Technische Universität Ilmenau

Fakultät für Informatik und Automatisierungstechnik

Diplomarbeit

Zum Thema

„Entwurf und Implementierung eines Regelungskonzeptes auf der Basis des Reinforcement Learning für einen Laborprozess.“

eingereicht von: Andreas Reuter
am: 01.06.2010
Hochschulbetreuer: Prof. Dr.-Ing. Steven Lambeck (JP)
Inventarisierungsnummer: 2010-03-01/022/II03/2215

URN: urn:nbn:de:gbv:ilm1-2010200212

Inhaltsverzeichnis

Kapitel 1
Einleitung

1.1 Motivation

Das Stabilisieren eines inversen Pendels gehört zu den klassischen Regelungsaufgaben. Immer dann, wenn in einem Prozess das Problem des Pendelns oder Balancierens einer Masse auftritt, kann dieses mit einem Pendel bzw. einem inversen Pendel modelliert werden. Die Anwendungsgebiete sind sehr vielfältig. Zu finden ist dieses Problem z. B. beim Simulieren des Laufens, bei der Stabilisierung und zielgerichteten Bewegung von Roboterarmen (doppelt inverses Pendel) oder bei Raketenantrieben. Auch beim Simulieren des Verhaltens einer Ladebrücke kann das System eines Pendels zur Modellierung herangezogen werden. Um dieses Standardproblem zu untersuchen, wurde zu Versuchs- und Praktikumszwecken ein entsprechender Versuchsaufbau im Vorfeld dieser Arbeit im Rahmen zweier Studienjahresarbeiten erstellt und bereits mit verschiedenen digitalen Regelalgorithmen betrieben. Bisher sind zum Stabilisieren des Pendels Standardregler genutzt worden, wie z. B. ein Zustandsregler (ZR) mit einigen Erweiterungen (ZR mit Vorfilter (VF), ZR mit PI-Vorregelung und VF). Mit diesen Ansätzen wurden z. T. gute Ergebnisse erreicht, trotzdem konnten einige durch den Versuchsaufbau bedingte Effekte nicht berücksichtigt werden. Dies soll mit dem Einsatz von künstlichen neuronalen Netzen zur Abbildung von Reinforcement Learning (RL) Algorithmen geändert werden. Ziel der Arbeit ist die Klärung der Frage, ob RL Algorithmen unter Verwendung künstlicher neuronaler Netze eine Verbesserung in der Regelung des inversen Pendels gegenüber den vorhandenen Algorithmen in Bezug auf verschiedene Ziele bringen können. Besonderer Wert wird auf die Beseitigung von beim Betrieb auftretenden Beeinflussungen durch Reibungseffekte gelegt. Zuerst muss jedoch überprüft werden, ob eine stabile Regelung mit den RL Algorithmen überhaupt möglich ist. Dazu werden im vorhandenen Versuchsaufbau neue Regelalgorithmen implementiert, diese mit verschiedenen Parametern getestet und die

gewonnenen Ergebnisse mit denen der oben genannten, bereits vorhandenen Algorithmen verglichen.

1.2 Aufgabenstellung

Zu Beginn der Arbeit steht eine Literaturrecherche zu den Themen künstliche neuronale Netze und Reinforcement Learning. Im Anschluss daran soll ein Überblick über die genannten Themen gegeben und die für die weitere Arbeit benötigten Grundlagen vermittelt werden. Außerdem ist ein kurzer Überblick über die bereits realisierten Regelalgorithmen zu geben. Die gewonnen Erkenntnisse sollen im Folgenden zur Konzeption geeigneter Algorithmen für den vorhandenen Versuchsaufbau genutzt werden. Diese Algorithmen sollen unter der Nutzung von MATLAB/Simulink entworfen und auf einer Rapid Control Prototyping (RCP) Plattform implementiert werden. Nach der Aufnahme der aus der Anwendung der klassischen und neuen Regelalgorithmen resultierenden Ergebnisse ist ein Vergleich zwischen diesen zu ziehen, um eventuelle Vor- und Nachteile beider Lösungen aufzuzeigen. Besonders ist dabei auf den Einfluss von Reibungseffekten zu achten.

1.3 Gliederung der Arbeit

Zunächst sollen die Voraussetzungen zum Lösen des gestellten Problems geschaffen werden. Dazu wird eine Zusammenfassung über Reinforcement Learning und künstliche neuronale Netze gegeben. Um die speziellen Bedingungen für die Anwendung zu verstehen, wird im Folgenden der Versuchsaufbau erklärt. Außerdem wird gezeigt, welche Regelalgorithmen bereits zur Regelung des Pendels genutzt werden. Dies geschieht, um im Weiteren einen Vergleich zwischen den herkömmlichen und den RL Algorithmen ziehen zu können. Im Anschluss daran werden die verschiedenen Ansätze zum Einsatz von RL entworfen und am Versuchsaufbau getestet. Die Ergebnisse aus diesen Versuchen werden mit denen aus den Versuchen mit den vorhandenen Regelalgorithmen verglichen und abschließend ein Fazit gezogen.

Kapitel 2
Grundlagen

2.1 Maschinelles Lernen

Maschinelle Lernverfahren werden angewendet, um aus bekanntem Wissen verschiedene Problemlösungen zu generieren und so intelligente Systeme zu schaffen. Die erstellten Systeme sollen unter anderem dazu in der Lage sein, vorhandenes Wissen korrekt abzubilden, von speziellem Wissen auf Allgemeines zu schließen (Generalisieren) oder auch zu klassifizieren. Die Anwendungsfelder für maschinelle Lernverfahren sind sehr verschieden und somit auch ihre Umsetzung. Eine Unterscheidung der Anwendungen kann beim Betrachten des repräsentierten Wissens getroffen werden. Man differenziert zwischen symbolischen und subsymbolischen Systemen, wobei zuerst Genannte das darzustellende Wissen explizit, also in Form von Regeln und Fakten aufzeigen. Subsymbolische Systeme speichern ihr Wissen auf indirekte Weise, es ist also nicht direkt einsehbar. Ein wichtiger Vertreter dieser Systeme sind die künstlichen neuronalen Netze (KNN).

Eine weitere Kategorisierung der Lernverfahren kann auf der algorithmischen Ebene vorgenommen werden. Sind die Ausgabedaten zu den gegebenen Eingabedaten bekannt, welche zur Bestimmung der Wissensrepräsentation verwendet werden, spricht man von überwachtem Lernen (Supervised Learning). Ein Experte stellt während des Lernens die korrekten Ausgabedaten zu den gegebenen Eingabedaten zur Verfügung.

Verfahren des unüberwachten Lernens (Unsupervised Learning) werden vor allem zur Klassifikation und zur Mustererkennung genutzt. Dazu werden Merkmale im Zustandsraum festgestellt und mit Hilfe dieser Objekte zu Klassen zugeordnet.

Dieses Kapitel widmet sich zunächst denjenigen Algorithmen des maschinellen Lernens, die ohne a priori-Informationen Wissen zu erlernen und zu repräsentieren versuchen, den sogenannten Algorithmen des bestärkenden Lernens (Reinforcement Learning). Sie werden immer dann angewendet, wenn keine expliziten Ein-/Ausgabedaten zur Verfügung stehen.

Man könnte Reinforcement Learning (RL) folgendermaßen übersetzen: zielgerichtetes Lernen durch Interaktion mit der Umwelt. (Vgl. [1], S.13)

Im Anschluss wird auf künstliche neuronale Netze eingegangen und eine Übersicht des im Rahmen dieser Arbeit benötigten Wissens zum Thema KNN gegeben.

2.2 Reinforcement Learning

Das bestärkende Lernen ist von den maschinellen Lernverfahren das dem „echten“ Lernen ähnlichste Verfahren. Die Grundlage natürlichen Lernens ist immer die Interaktion eines „intelligenten“ Systems, also eines Systems, das aus den eingehenden Informationen Schlüsse ziehen kann, aus der es umgebenden Umwelt (über Sensoren). Im Allgemeinen reagiert der Mensch, nachdem er sich in Relation zu seiner Umwelt gesetzt hat, also den ihn umgebenden Zustandsraum ermittelt hat, mit einer Aktion, welche wiederum eine Änderung in der Umwelt hervorruft. Diese Änderung wird vom Menschen dahingehend interpretiert, ob sie den gewünschten Effekt bezüglich seiner vorher festgelegten Ziele erreicht hat. Der Erfolg oder Misserfolg der getätigten Aktion wird abgeschätzt und das zukünftige Verhalten in gleichen bzw. ähnlichen Situation angepasst. Dieses Verhalten kann nicht nur beim Menschen beobachtet werden sondern bei allen intelligenten Lebewesen. Das Prinzip des Lernens durch Interaktion mit der Umwelt und die Interpretation der Ergebnisse ist ein wichtiger Bestandteil unserer Definitionen von Intelligenz und Evolution.

Reinforcement Learning (RL) bedeutet in diesem Kontext, den Versuch zu unternehmen, die allgemeinen Lernabläufe auf mathematische Algorithmen abzubilden und so für ingenieurwissenschaftliche Probleme nutzbar zu machen.

Eines der Hauptprobleme beim RL besteht zweifelsohne im Finden der richtigen Taktik zur Problemlösung. Erreicht der Algorithmus sein Ziel, erfährt er eine Belohnung (Reward). Beim Erreichen nicht gewollter Zustände wird ein negatives Signal an das lernende System zurückgekoppelt. Auf diese Weise findet das Lernen statt. Primärziel des Algorithmus ist die Maximierung der erhaltenen Belohnung über den gesamten Zeithorizont. Treten Erfolg oder Misserfolg erst nach einer Serie von Handlungen auf, kann die Richtigkeit der einzelnen

Aktionen nicht sofort abgeschätzt werden. Erst beim Erreichen des gestellten Ziels ist eine Evaluierung der gewählten Taktik möglich. Es stellt sich jedoch die Frage, welche der getätigten Handlungen einen Einfluss auf die Erreichung des Ziels hatten und wie groß dieser jeweils war. Dieses Problem kann mit dem Einführen einer Funktion zur Abschätzung der Güte einer getätigten Aktion gelöst werden (Wertefunktion). Für Alpaydin ist diese Funktion ein „interner Belohnungsmechanismus“, welcher „(…) ausdrückt, wie gut sie [die Aktionen] uns in Richtung Ziel weiterbringen und uns zur wahren Belohnung führen.“ ([1] S.398). Weiterhin ist zu klären, ob die gewählte Taktik die optimale ist, ob sie tatsächlich nicht nur zum Erfolg sondern zum maximalen Erfolg führt. Dazu ist ein „Weiterlernen“ des Systems, ein Erforschen (Exploration) des (unbekannten) Zustandsraums durch die Wahl verschiedener Aktionen bei gleichen Zuständen von Nöten. Um das gestellte Ziel trotz Weiterlernens noch erreichen zu können, ist es wichtig, eine gute Balance zwischen dem Erforschen neuer Zustände und dem Ausnutzen bereits erlernter, guter Aktionen (Exploitation) zu finden.

Im folgenden Abschnitt werden die zur Realisierung solcher Systeme benötigten Komponenten genannt und kurz charakterisiert. Es wird weiterhin ein Verständnis für die Begriffe Wertefunktion und Optimalität im Kontext des Reinforcement Learning geschaffen. Anschließend wird aufgezeigt, wie das Lernen realisiert werden kann, wenn das Umweltmodell bekannt ist. Welche Vor- und Nachteile das Erforschen bzw. Ausbeuten (un-) bekannter Aktionen mit sich bringt, soll im Anschluss beleuchtet werden. Dazu werden die Begriffe temporale Differenz und Q-Learning erläutert. Es wird weiterhin ein Überblick darüber gegeben, wie das Umweltmodell gespeichert wird und was der Begriff Generalisierung bedeutet. Die abschließende Betrachtung der Actor-Critic-Methoden gibt einen Überblick darüber, wie RL für Regelungen eingesetzt werden kann.

2.2.1 Eigenschaften eines Reinforcement Learning Systems

Den lernenden Teil eines Reinforcement Learning System (RLS), sei es ein Roboter bzw. die ihn steuernden Algorithmen oder ein selbstlernendes Computerprogramm, das auf irgendeine Art auf Situationen reagieren muss (akustische Anfragen, Steuern eines

Industrieprozesses etc.), bezeichnet man als **Agenten**. Dieser ist über Sensoren mit seiner **Umwelt** verbunden. Der Agent hat ein **Ziel**, das er mit Hilfe einer zu erlernenden **Taktik** erreichen will. Das Ziel definiert sich durch die zu lösenden Aufgabe. Vorstellbar ist z. B. das möglichst stromsparende Einsammeln von Müll durch einen Roboter oder das Gewinnen eines Brettspiels mit der minimalen Anzahl von Zügen. Der Agent beobachtet mit Hilfe seiner Sensoren die Umgebung (Umwelt), um deren **Zustände** abzuschätzen und sich in Relation zu ihr setzen zu können. Er fällt aufgrund seiner Beobachtungen Entscheidungen, welche zu Aktionen seinerseits führen (Interaktion mit der Umwelt). Diese Aktionen rufen zum einen eine Veränderung der Zustände der Umwelt und zum anderen eine **Rückmeldung** als Belohnung bzw. Bestrafung ausgehend von der Umwelt hervor, welche wiederum vom Agenten zum Abschätzen der Güte der getätigten Aktion hinsichtlich des zu erreichenden Ziels genutzt werden. Die wichtigste Aufgabe des Agenten besteht darin, diese Belohnung über den gegebenen Zeithorizont zu maximieren.

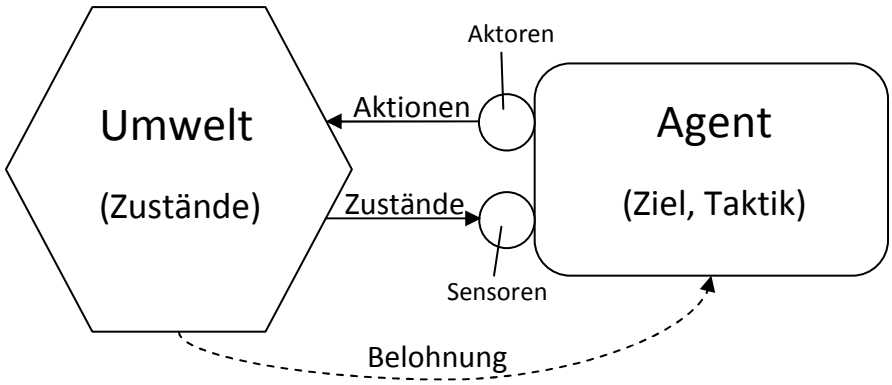


Abbildung 1: Schema eines RLS

Der betrachtete Zeithorizont ist diskret, d. h. $t = 0,1,2,\ldots$ und die Zustände entstammen dem Zustandsraum Z, wobei der Zustand zum Zeitpunkt t mit z_t und der Folgezustand mit z_{t+1} usw. bezeichnet werden. Zum Zeitpunkt t liegt ein Zustand z_t vor, auf den der Agent mit einer Aktion a_t aus dem Aktionsraum $A(z_t)$ reagiert. Resultierend daraus sind der Folgezustand z_{t+1} und die empfangene Belohnung r_{t+1}.

Die von einem Agenten befolgte **Taktik** bezeichnet man mit π. Sie trifft eine Aussage darüber, welche Aktion in welchem Zustand ausgeführt wird: $a_t = \pi(z_t)$. Es findet also eine Abbildung der Zustände auf die Aktionen statt. Um die Güte einer solchen Taktik zu

bestimmen, wird der **Wert** $V^{\pi}(z_t)$ einer Taktik π, d. h. die über den gesamten Zeithorizont aufsummierte zu erwartende Belohnung beim Befolgen der Taktik, folgendermaßen bestimmt:

$$V^{\pi}(z_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]. \tag{2.1}$$

Diese Gleichung findet jedoch nur dann Anwendung, wenn ein finiter Zeithorizont betrachtet wird, der Agent also versucht, die zu erwartende Belohnung für die nächsten, endlich vielen T Zeitschritte beim Befolgen der Taktik zu berechnen (vgl. [1], S.401).

Betrachtet man einen infiniten Zeithorizont, werden alle in der Zukunft liegenden, erwarteten Belohnungen mit in Betracht gezogen. Sie werden je nach Wunsch diskontiert, also mit einem Faktor $0 \leq \gamma \leq 1$ gewichtet. Wählt man $\gamma = 0$, geht nur die sofortige Belohnung in die Berechnung mit ein. Um alle in der Zukunft liegenden Belohnungen mit einzubeziehen, wählt man γ nahe 1. Ein Wert von $\gamma > 1$ ist denkbar, aber nicht interessant, da die meisten Aufgabenstellungen in einem zeitlichen Limit bewältigt werden sollen und somit die Belohnungen, die in der nahen Zukunft liegen, wichtiger sind und deshalb auch stärker gewichtet werden sollten, als die späteren. Die Formel für den Wert einer Taktik verändert sich durch die infinite Betrachtung wie folgt (vgl. [1], S.402):

$$V^{\pi}(z_t) = E[r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i}\right]. \tag{2.2}$$

Außer dem Wert einer Taktik, $V^{\pi}(z_t)$ sind weitere Wertebetrachtungen möglich. Der Wert eines Zustands wird mit $V(z_t)$ angegeben und liefert eine Aussage darüber, wie günstig es für den Agenten ist, sich im Zustand z_t zu befinden (vgl. [1], S.402). Ist es von Interesse, wie vorteilhaft die Wahl der Aktion a_t im Zustand z_t ist, spricht man vom Wert des **Zustands-Aktions-Paares** $Q(z_t, a_t)$. Das Bestimmen von direkten Belohnungen ist relativ einfach, wohingegen das Bestimmen des Wertes eines Zustandes sehr viel komplizierter sein kann. Der Aufwand zur Berechnung lohnt sich jedoch, da der Wert eines Zustands eine Aussage darüber trifft, wie groß die folgende Gesamtbelohnung ist und nicht nur die direkt auf die Aktion Folgende (vgl. [2], S.8).

2.2.2 Wertefunktionen und Optimalität

Die vom Agenten gewählte Taktik soll die zu erwartende Belohnung über den betrachteten Zeithorizont maximieren. Dies kann nur mit einer optimalen, im Laufe der Lernphase zu bestimmenden Taktik π^* geschehen. Es gilt (vgl. [1], S.402):

$$V^*(z_t) = \max_{\pi} V^{\pi}(z_t), \forall z_t . \tag{2.3}$$

Das bedeutet, dass der beste Wert für einen Zustand $V^*(z_t)$ durch diejenige Taktik bestimmt ist, deren Ausführung über den gesamten Zeithorizont betrachtet und für alle möglichen Zustände die maximale Belohnung zur Folge hat. Ausgehend von der Zustands-Aktions-Paare Wertbetrachtung $Q(z_t, a_t)$ kann folgende Aussage getroffen werden: $Q^*(z_t, a_t)$ bezeichnet die aufsummierte Belohnung, die der Aktion a_t folgt, wenn sie im Zustand z_t ausgeführt wird und anschließend die optimale Taktik gefahren wird. Der größte Wert eines Zustandes entspricht dann dem Wert der bestmöglichen Aktion (vgl. [1], S.402).

$$V^*(z_t) = \max_{a_t} Q^*(z_t, a_t) \tag{2.4}$$

$$V^*(z_t) = \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i}\right] \tag{2.5}$$

$$V^*(z_t) = \max_{a_t} E\left[r_{t+1} + \gamma \cdot \sum_{i=1}^{\infty} \gamma^{i-1} \cdot r_{t+i+1}\right] \tag{2.6}$$

$$V^*(z_t) = \max_{a_t} E[r_{t+1} + \gamma \cdot V^*(z_{t+1})] \tag{2.7}$$

Betrachtet man die Schaltwahrscheinlichkeiten in den nächsten Zustand $P(z_{t+1}|z_t, a_t)$, also die Wahrscheinlichkeit mit der ein Zustand in den nächsten übergeht, ändert sich die Wertefunktion folgendermaßen (Bellmannsche Gleichung) (vgl. [1], S.402):

$$V^*(z_t) = \max_{a_t}\left(E[r_{t+1}] + \gamma \cdot \sum_{z_{t+1}} P(z_{t+1}|z_t, a_t) \cdot V^*(z_{t+1})\right). \tag{2.8}$$

Zunächst wird im Zeitpunkt t eine Aktion a_t gewählt und die direkt folgende Belohnung r_{t+1} berechnet. Mit einer Wahrscheinlichkeit von $P(z_{t+1}|z_t, a_t)$ geht das System nun in den Folgezustand z_{t+1} über. Wird die optimale Taktik weiterhin befolgt, kann der Wert der

nachfolgenden Belohnung mit $V^*(z_{t+1})$ bezeichnet werden. Da jedoch mehrere Folgezustände schalten können, muss jede mögliche folgende Belohnung mit den korrespondierenden Schaltwahrscheinlichkeiten multipliziert, alle Werte aufsummiert und, da sie in der Zukunft liegen, mit γ diskontiert werden. Dieser Wert wird zur direkten Belohnung r_{t+1} addiert und festgehalten. Nachdem alle möglichen Werte für alle möglichen Aktionen a_t berechnet wurden, wird diejenige Aktion a_t^* als die optimale betrachtet, welche den Maximalwert der Berechnung ergeben hat. Dieser Wert entspricht der maximal zu erreichenden Gesamtbelohnung.

Eine Betrachtung der Notation für die Zustands-Aktions-Paare ergibt (vgl. [1], S.403):

$$Q^*(z_t, a_t) = E[r_{t+1}] + \gamma \cdot \sum_{z_{t+1}} P(z_{t+1}|z_t, a_t) \cdot \max_{a_{t+1}} Q^*(z_{t+1}, a_{t+1}) \,. \tag{2.9}$$

Zum Zeitpunkt t ist also diejenige Aktion a_t optimal, welche die Gleichung

$$Q^*(z_t, a_t^*) = \max_{a_t} Q^*(z_t, a_t) \tag{2.10}$$

erfüllt. Die optimale Taktik für einen Schritt ist also diejenige, welche die Aktion a_t^* wählt. Um die optimale Taktik für den gesamten Ablauf zu bestimmen, müssen Informationen zu den Schaltwahrscheinlichkeiten und den Wahrscheinlichkeiten zum Auftreten einer Belohnung in einem bestimmten Zustand bekannt sein. Ein sehr genaues Modell zu der den Agenten umgebenden Umwelt muss also vorliegen. Ist diese Vorrausetzung erfüllt, kann die Berechnung der Wertefunktion und der sich daraus ergebenden optimalen Taktik z. B. mittels dynamischer Programmierung durchgeführt werden. Ist dieses Wissen nicht bekannt, muss die Umwelt erforscht und ein Modell für diese gebildet werden (vgl. [1], S.403f).

2.2.3 Lernen bei bekannten Belohnungen und Schaltwahrscheinlichkeiten

Im deterministischen Fall kann mittels **Wertiteration**, einem Algorithmus, der zu den optimalen Zustandswerten konvergiert, die optimale Wertefunktion gefunden werden. Da alle Schaltwahrscheinlichkeiten und Belohnungen bekannt sind, kann diese Funktion wie folgt berechnet werden: zunächst müssen alle Werte $V(z)$ willkürlich initialisiert werden.

Anschließend werden für alle Zustände z aus Z die möglichen Aktionen betrachtet und für diese die Werte $Q(z_t, a_t)$ mit

$$Q(z_t, a_t) = E[r_{t+1}] + \gamma \cdot \sum_{z_{t+1}} P(z_{t+1}|z_t, a_t) \cdot V(z_{t+1}) \tag{2.11}$$

berechnet (vgl. [1], S.404). Der Maximalwert $V(z) = \max_a Q(z, a)$ wird für jeden Zustand festgelegt und dient als neuer Zustandswert. Dieser Ablauf wird solange wiederholt, bis die Werte $V(z)$ konvergieren, also genau so lang, bis die größte Wertedifferenz aller Werte $V(z)$ einer Iteration im Vergleich zur vorherigen Iteration, also $\max_{z \in Z} |V^{(l+1)}(z) - V^l(z)|$ (l : Zählvariable der Iterationen) unter einen Schwellwert δ fällt.

Eine weitere Möglichkeit, um bei bekannten Belohnungen und Schaltwahrscheinlichkeiten die optimale Taktik zu finden, besteht im Anwenden der **Taktikiteration**. Bei dieser werden alle Werte $V(z)$ durch eine Taktik berechnet und im Anschluss die Taktik verbessert. Tritt keine Verbesserung mehr ein, ist die optimale Taktik gefunden. Die optimale Taktik wird so in einer geringeren Anzahl an Iterationen als bei der Wertiteration gefunden, wobei die Berechnung der einzelnen Iterationen mehr Zeit in Anspruch nimmt (vgl. [1], S.404f).

2.2.4 Lernen durch Exploration

Die **Exploration**, also die Erkundung des Aktionsraumes und somit neuer Zustandswerte, ist immer dann nötig und sinnvoll, wenn das Umgebungsmodell nicht genau bekannt ist. In diesem Fall sind die Schaltwahrscheinlichkeiten der Folgezustände und die folgenden Belohnungen aufgrund unzureichender a priori-Informationen über die Umwelt gar nicht oder nur teilweise bekannt und ein Modell der Umgebung muss angelernt werden. Um dies zu realisieren, wird die Differenz zwischen dem aktuellen Wert und der Summe aus der folgenden Belohnung und dem diskontierten Folgewert untersucht und zur Aktualisierung des aktuellen Wertes genutzt. Der durch die zeitliche Verschiebung entstehende Unterschied gab Verfahren dieser Art den Namen „Algorithmen mit temporaler Differenz“. Der Begriff bezieht sich dabei auf die Aktualisierung der jeweiligen betrachteten Werte $V(z)$ bzw. $Q(z, a)$.

Beim Erkunden werden in den wenigsten Fällen die besten Aktionen sofort gewählt. Dementsprechend ist auch die folgende Belohnung nicht immer die maximal Mögliche. Um auch während des Explorierens dem Gesamtziel näher zu kommen und um die Exploration nicht unendlich lang andauern zu lassen, empfiehlt es sich mit einer gewissen Wahrscheinlichkeit keine Erforschung, sondern eine sogenannte **Exploitation** durchzuführen, also eine Ausnutzung der besten bekannten Aktion. Ein Beispiel für ein Verfahren dieser Art bietet die ***ϵ*-Greedy-Suche**. Bei ihr wird mit einer Wahrscheinlichkeit von ϵ eine zufällige Aktion aus allen möglichen Aktionen und mit einer Wahrscheinlichkeit von $1 - \epsilon$ die beste bisher bekannte Aktion gewählt.

Die Auswahl der Aktionen kann auch von den Werten der Zustände $V(z)$ bzw. der Zustands-Aktions-Paare $Q(z, a)$ abhängig gemacht werden. Die Gleichung

$$P(a|z) = \frac{e^{[Q(z,a)/T]}}{\sum_{b \in A}^{A} e^{[Q(z,b)/T]}} \tag{2.12}$$

gibt ein Beispiel für dieses Prinzip (vgl. [1], S.406). Die Variable T beeinflusst hier die Exploration/Exploitation. Zu Beginn des Lernprozesses sollte diese groß gewählt werden, da die Werte der Zustands-Aktions-Paare somit kaum eine Rolle spielen. Reduziert man T, haben die Werte $Q(z, a)$ einen größeren Einfluss auf die Wahl der Aktionen. Aktionen mit großen Werten werden häufiger gewählt.

Erst nachdem die Wertefunktion hinreichend gut geschätzt wurde, kann der Agent die optimalen Aktionen in den jeweiligen Zuständen prognostizieren. Die Werte der einzelnen Zustände sind vor der Exploration meist unbekannt. Deshalb wird zu Beginn des Lernens allen Zustandswerten bzw. den Werten der Zustands-Aktions-Paare der gleiche Wert zugewiesen (z. B. $Q(z, a) = 0, \forall z$). Die Aktualisierung dieser Werte erfolgt immer nach der Beobachtung eines Zeitschrittes. Die direkt folgende Belohnung r_{t+1} und der Wert $Q(z_{t+1}, a_{t+1})$ dienen als Grundlage zur Aktualisierung des neuen Wertes von $Q(z_t, a_t)$. Existiert nicht nur eine mögliche Folgeaktion und Belohnung, gilt (vgl. [1], S.408f):

$$Q^{neu}(z_t, a_t) = Q^{alt}(z_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_{a_{t+1}} Q(z_{t+1}, a_{t+1}) - Q^{alt}(z_t, a_t) \right). \tag{2.13}$$

Jedes auftretende Zustands-Aktions-Paar wird beobachtet und der neue Wert der vorherigen Aktion berechnet. Die Iteration ist beendet, sobald der Zielzustand erreicht

wurde. Dieser Ablauf ist als **Q-Lernalgorithmus** bekannt. Die Konvergenz des Verfahrens wurde von Watkins und Dayan nachgewiesen ([10]).

Die Zustandswerte werden korrespondierend dazu folgendermaßen aktualisiert:

$$V^{neu}(z_t) = V^{alt}(z_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot V(z_{t+1}) - V^{alt}(z_t)\right). \quad (2.14)$$

Dieses Verfahren trägt den Namen **TD-Lernen** (Sutton 1988) (vgl. [1], S.410).

Im Prinzip stellt $r_{t+1} + \gamma \cdot V(z_{t+1})$ die spätere Schätzung des Zustandes $V(z_t)$ dar (vgl. Kapitel 2.2.2) und der Unterschied zwischen beiden ist die **temporale Differenz** (**TD**). Durch eine schrittweise Verringerung des Aktualisierungsfaktors α kann im Laufe des Lernens die Genauigkeit des Algorithmus verbessert werden. Das Verfahren konvergiert garantiert zur optimalen Wertefunktion $V^*(z)$ (vgl. [1], S. 410).

2.2.5 Generalisierung und Speicherung des Umweltmodels

„Generalisierung ist der Prozess, eine Beschreibung des Ganzen aus einzelnen Teilen abzuleiten, vom konkreten zum allgemeinen Fall zu schließen oder aus dem Vorhandensein einer oder mehrerer Instanzen eine Klasse von Objekten zu definieren.“ ([3], S.38) Ist ein System in der Lage zu **Generalisieren**, kann es aus bekanntem Wissen neues Wissen ableiten. Ein Beispiel dafür sind Verfahren der Mustererkennung. Einem intelligenten System werden Beispiele eines Musters vorgelegt und das System erlernt die Eigenschaften des Musters, nicht die exakte Abbildung dieser. Auf diese Weise kann später ein dem System unbekanntes Beispiel des gleichen Musters, auch wenn es bei der Eingabe verrauscht ist, erkannt und zugeordnet werden. Ein gutes Beispiel für Systeme, die in der Lage sind zu Generalisieren, sind künstliche neuronale Netze (KNN). Ein weiterer Vorteil der KNN besteht in ihrer Eigenschaft, ihr repräsentiertes Wissen implizit und dadurch komprimiert zu speichern. Bei kleinen Aktions- und Zustandsräumen können die bei der Findung der optimalen Wertefunktion errechneten Werte in einer Tabelle abgelegt werden. Man spricht von tabularen Algorithmen (vgl. [1], S. 412). Existieren viele mögliche Zustände und Aktionen, wird eine andere Art der Datenspeicherung notwendig. Hier bietet sich die

Verwendung von KNN zur Speicherung der Taktik aufgrund der zuvor aufgezeigten Eigenschaft zur „Datenkompression“ an.

2.2.6 Actor-Critic-Methoden

Um die oben genannten Algorithmen und Methoden des RL effektiv für Regelungsprobleme nutzen zu können, empfiehlt sich die Verwendung von **Actor-Critic-Methoden**. Bei diesen wird eine explizite Trennung zwischen Generierung/Darstellung der Taktik und der Wertefunktion eines RLS vorgenommen. Der „Actor“ repräsentiert hierbei die Taktik, also die in der Regelung getätigten Aktionen am Stellglied. Diese Aktionen werden durch die Wertefunktion auf ihre Güte hin untersucht. Dies geschieht durch einen Vergleich der Werte zweier aufeinanderfolgender Zustände (nach getätigten Aktionen). Die Differenz dieser beiden Werte stellt die in Kapitel 2.2.4 vorgestellte temporale Differenz dar.

$$\delta_t = r_{t+1} + \gamma * V(z_{t+1}) - V(z_t) \tag{2.15}$$

Ein positiver Wert δ_t weist darauf hin, dass ein Zustandsübergang mit einer Wertsteigerung stattgefunden hat. Die zum jeweiligen Zustand gewählte Aktion am Stellglied sollte also bevorzugt oder sogar erhöht werden. Ein negativer Wert weist demnach auf eine schlechte Aktion hin, aufgrund derer sich Zustände mit einem kleineren Wert eingestellt haben (vgl. [2], S.151ff.).

Beim Verwenden dieser Methode ist darauf zu achten, dass zunächst die Wertefunktion angelernt werden muss, um eine qualifizierte Aussage über die getätigten Aktionen treffen zu können. Abbildung 6, S.32 zeigt den schematischen Aufbau eines RLS, das mit der Actor-Critc-Methode arbeitet.

2.3 Künstliche Neuronale Netze

Die Reizweiterleitung und -verarbeitung spielt eine sehr wichtige Rolle in der Natur. Die Mechanismen, die diese Abläufe realisieren, sind im Laufe der Evolution entstanden und ermöglichen es den Lebewesen auf sehr effiziente Weise Informationen zu übermitteln und zu speichern. Mechanische Reize, wahrgenommen durch Nervenzellen auf der Haut eines Menschen, werden in kürzester Zeit zu seinem Gehirn weitergeleitet, dort verarbeitet und eine entsprechende Reaktion ausgelöst. Die gleichzeitig zu bewältigende Aufgabenvielfalt ist dabei sehr groß und kann trotzdem erfüllt werden. Grund dafür ist die massive parallele Verarbeitung der ankommenden Reize im Gehirn (vgl. [3], S.13).

Es liegt nahe, den Versuch zu unternehmen, diese Mechanismen nachzubilden und Lernverfahren zu entwickeln, die adaptive Lösungen liefern können. Der Algorithmus soll in der Lage sein, Wissen dynamisch zu repräsentieren und aus dem Erlernten intelligente Lösungen für die spezifischen Aufgabenstellungen zu generieren. Um diesen Nachbau zu realisieren ist es hilfreich, sich den Aufbau und die Funktionsweise derjenigen Zellen vor Augen zu führen, die für die Weiterleitung und Verarbeitung der Reize in der Natur verantwortlich sind. Die ersten Untersuchungen zu diesem Thema wurden von Warren McCulloch und Walter Pitts durchgeführt und resultierten in einem Formalmodell für das Neuron (vgl. [5], S.16). Seitdem wurden viele Anwendungen zu künstlichen neuronalen Netzen erforscht und präsentiert.

Typische Anwendungsgebiete für KNN sind Mustererkennung, Klassifizierung, Datenkomprimierung, Modellierung und Vorhersage, Optimierung, adaptive Steuerungen, Assoziatives Lernen, Kombinatorische Problemlösung etc. Diese Anwendungen können vor allem durch die folgenden Punkte realisiert werden: die Fähigkeit künstlicher neuronaler Netze zu generalisieren, zur parallelen Verarbeitung und zum adaptiven Lernen. Außerdem zeichnen sie sich durch eine hohe Fehlertoleranz aus und es hat sich gezeigt, dass sie auch bei unvollständigen oder verrauschten Input-Daten gute Ergebnisse erzielen können (vgl. [3], S. 14).

Der folgende Abschnitt beschreibt den Aufbau und die Funktionsweise der in KNN verwendeten Neuronen. Anschließend wird das Perceptron als Spezialfall eines KNN

dargestellt und erläutert. Weiterhin werden die wichtigsten Lernverfahren kurz vorgestellt und ihre Vor- und Nachteile aufgezeigt.

2.3.1 Aufbau und Struktur

Die grundlegende Einheit, die jedes KNN besitzt, ist das Neuron. Jedes Neuron ist gekennzeichnet durch seinen Input, seinen Aktivitätslevel und seinen Output. Der Input wird durch die **Propagierungsfunktion** bestimmt und fasst alle an dem Neuron anliegenden Eingänge und die zugehörigen Gewichte zusammen. Häufig ist die Propagierungsfunktion als Linearkombination von Eingängen und Gewichten, also $netinput_i = \sum_j y_j w_{ij}$ definiert. Das bedeutet, alle Eingänge (bzw. Ausgänge aus der vorherigen Schicht) y_j in das empfangende Neuron i werden mit ihren korrespondierenden Gewichten w_{ij} multipliziert und anschließend aufsummiert. Der Index j steht dabei für das sendende Neuron. Geometrisch kann diese Funktion auch als Skalarprodukt interpretiert werden. Weitere Propagierungsfunktionen können als Distanz dargestellt werden und sind z. B. Euklidische Distanz, Maximum Distanz, Minimum Distanz und Mahalanobis Distanz.

Der berechnete Gesamtinput in das Neuron wird über eine **Aktivierungsfunktion** einem **Aktivitätslevel** a_i zugeordnet. Die Wahl dieser Funktion spielt vor allem beim Lernen im Netz eine bedeutende Rolle. Einige Lernverfahren sind auf Gradientenbildung angewiesen. Verwendet man eine nicht stetige bzw. nicht differenzierbare Aktivierungsfunktion, muss dies bei der Anwendung der Lernverfahren beachtet werden. Abbildung 2 zeigt häufig verwendete Aktivierungsfunktionen. Die einzelnen Ansätze bieten einige Vor- und Nachteile. Eine Begrenzung des Aktivitätslevels beispielsweise ist in vielen Anwendungsfällen sinnvoll und dem natürlichen Vorbild ähnlicher als ein linear unendlich steigender Aktivitätslevel (vgl. [5], S.19ff.).

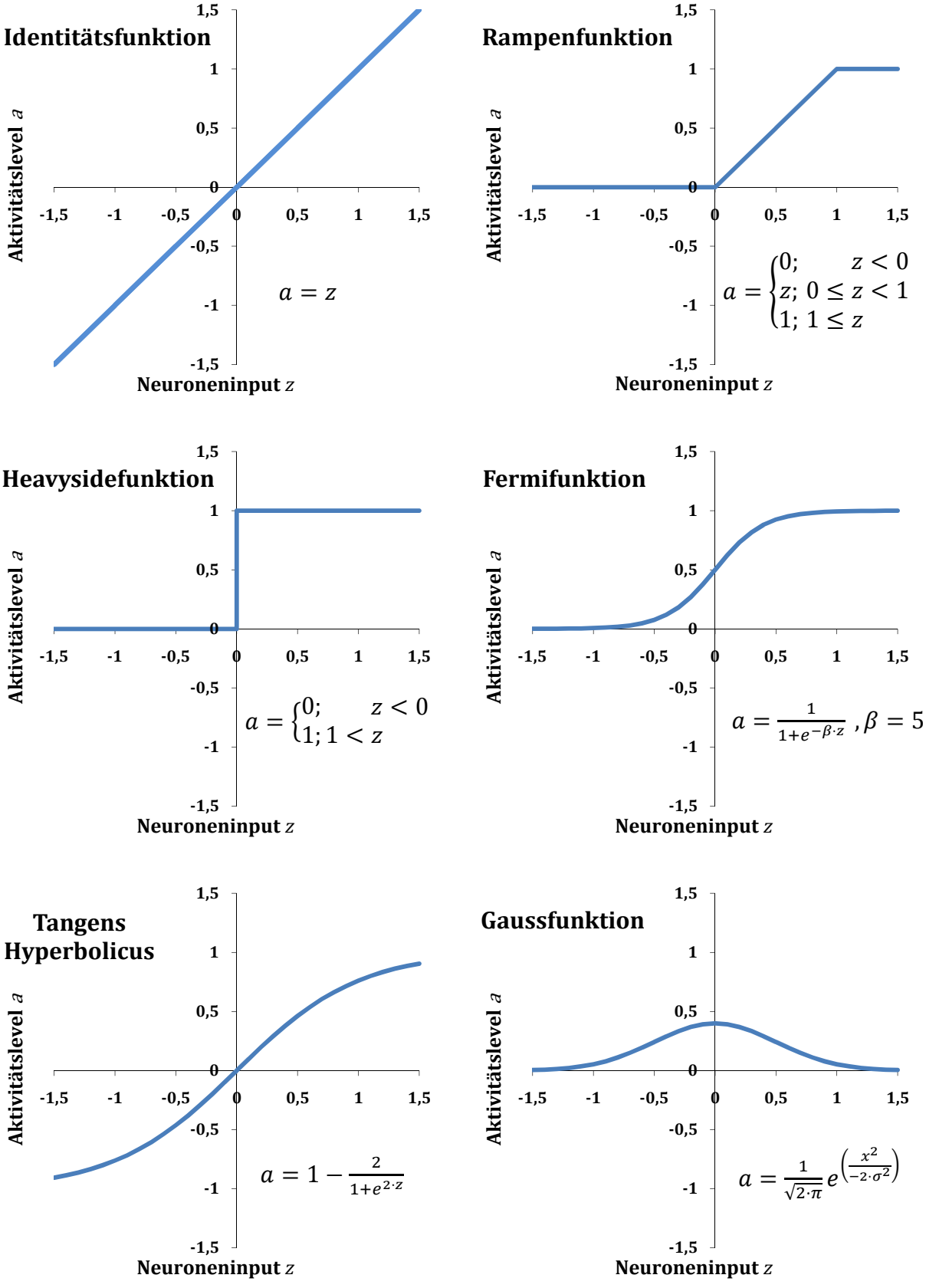


Abbildung 2: Einige in der Praxis verwendete Aktivierungsfunktionen (vgl. [5], S.23)

Der **Output** des Neurons i, y_i, wird in den meisten Fällen über die Identitätsfunktion gebildet, d. h. $a_i = y_i$. Im Folgenden wird der Output eines Neurons immer mit seiner Aktivierung gleichgesetzt. Die berechnete Ausgabe wird auf alle Eingänge der verbundenen Neuronen der folgenden Schicht aufgeschaltet, es findet also keine Aufteilung statt (vgl. [5], S.28).

Es gibt grundsätzlich vier verschiedene Arten von Units (die Begriffe „Unit“ und „Neuron“ werden synonym verwendet), abhängig davon, in welche Schicht sie eingeordnet sind: **Input-Units**, welche mit dem jeweiligen Umweltreiz (in Form von Zahlenwerten) beaufschlagt werden; **Hidden-Units**, die sich zwischen den Input- und den Output-Units befinden; **Output-Units**, welche den Ausgang des KNN darstellen und **Bias-Units**. Letztere können an beliebiger Stelle im KNN eingesetzt werden, haben einen konstanten Output und stellen auf diese Weise abhängig davon, wie die Gewichte zwischen Bias- und verbundenen Neuronen gewählt sind, eine Voraktivierung (positive Gewichte) bzw. eine Hemmung (negative Gewichte) der beeinflussten Neuronen sicher. Die Gewichte zwischen den Bias- und den mit ihnen verbunden Neuronen realisieren auf diese Weise einen Schwellwert für die betreffende Unit.

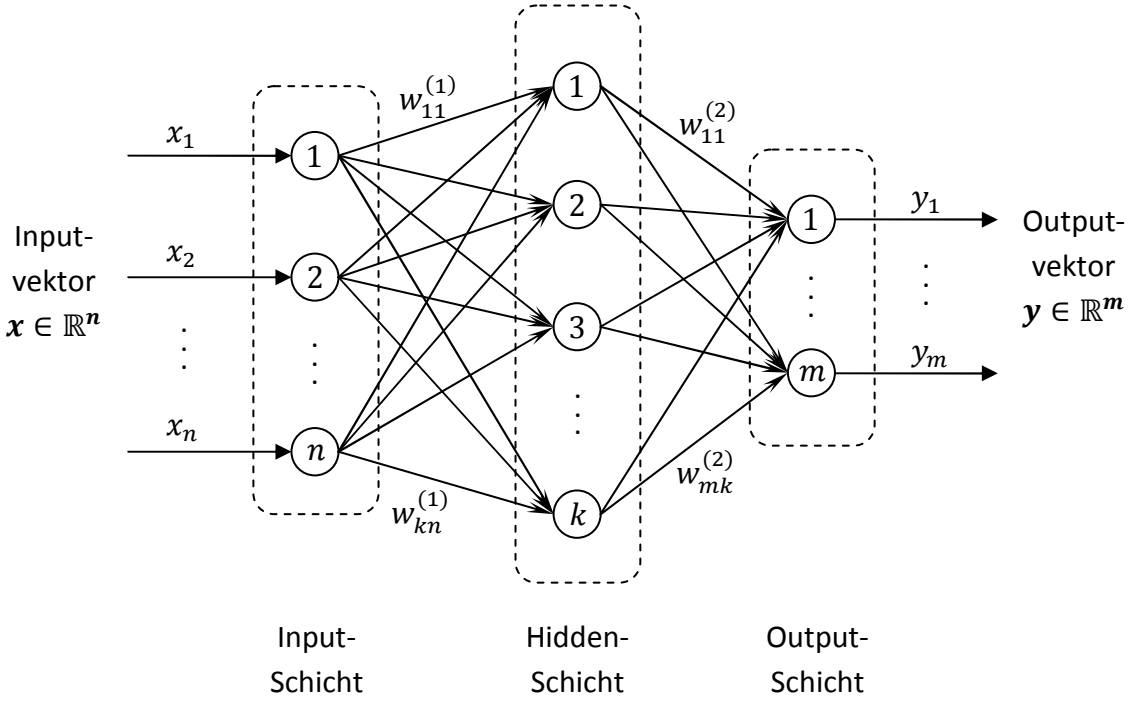


Abbildung 3: Aufbau künstlicher neuronaler Netze

Die Neuronen eines Netzes sind (teilweise) untereinander verknüpft, was die Weiterleitung des Inputreizes sicherstellt. Neuronen in derselben Schicht sind im Allgemeinen nicht miteinander verknüpft. Abbildung 3 zeigt einen Aufbau eines KNN und stellt die Beziehungen der Schichten und der zugehörigen Neuronen zueinander dar.

Es ist zu erkennen, dass alle Neuronen der Vorgängerschicht (die Schichten werden mit einem hochgestellten Index an den Gewichten gekennzeichnet) mit der nachfolgenden verbunden sind und dass alle Verbindungen in Richtung Output zeigen. Diese Art Netz bezeichnet man als **vollvermascht** und **nicht rekurrent**. Rekurrente Netze erkennt man daran, dass die Verbindungen zwischen den Neuronen auch vom Output zum Input gerichtet sein können, wodurch Schleifen entstehen. Jede Verbindung bzw. Kante stellt eine gewichtete Weiterleitung des Outputs des vorangegangen Neurons dar. Diese Gewichtung wird numerisch im Allgemeinen durch einen Zahlenwert zwischen -1 und 1 dargestellt. Negative Gewichte haben eine hemmende (inhibitorische), positive Gewichte eine bestärkende (exzitatorische) Wirkung auf nachgeschaltete Neuronen. In den meisten Fällen werden diese Gewichte beim Lernen adaptiert und dadurch die gewünschte Abbildung realisiert. Sie repräsentieren das im Netz gespeicherte Wissen. Die erste im tiefgestellten Index des Gewichtes verwendete Zahl repräsentiert das Empfänger-Neuron, die zweite das Sender-Neuron. Der hochgestellte Index gibt Auskunft darüber, in welcher Schicht sich das Empfängerneuron befindet. Da den Neuronen der Input-Schicht keine Gewichte vorgeschaltet sind, bezeichnet man die der Input-Schicht Folgenden als Gewichte der ersten Schicht. Die Gesamtheit der Gewichte kann in der Gewichtsmatrix $\boldsymbol{W}$ zusammengefasst werden, wobei die Gewichte der einzelnen Schichten spaltenweise aufgeschrieben werden ($\boldsymbol{W} = [\boldsymbol{W_1}, \boldsymbol{W_2}, \dots, \boldsymbol{W_n}]$; n - Anzahl der Schichten). $\boldsymbol{W_1}$ enthält also alle Gewichte der ersten Schicht $w_{11}^{(1)}, \dots, w_{kn}^{(1)}$.

Ein neuronales Netz kann keine, eine oder mehrere Hiddenschichten besitzen, wobei Hornik, Stinchcombe und White (1989) gezeigt haben, dass alle Probleme, die mit mehreren Hiddenschichten gelöst wurden, bei der Wahl genügend vieler Hidden-Neuronen auch mit einer Hiddenschicht gelöst werden können (vgl. [5], S.17f).

2.3.2 Wissensabbildung

Wie bereits erwähnt, erfolgt die Abbildung des gespeicherten Wissens im Allgemeinen in den Gewichten. Ein n-dimensionaler Eingangsvektor wird also, abhängig von den zugehörigen Gewichten und den verwendeten Propagierungs- und Aktivierungsfunktionen, auf einen m-dimensionalen Ausgangsvektor abgebildet (vgl. [3], S.29). Mathematisch lässt sich diese Aussage folgendermaßen darstellen:

$$y = f(x, g). \tag{2.16}$$

Bei der Verwendung nur einer Schicht im Netz können keine nichtlinearen Funktionen abgebildet werden, was die Einsatzmöglichkeiten der KNN stark einschränkt. Die Verwendung einer (oder mehrerer) Hiddenschicht(en) löst dieses Problem.

Auch die Wahl der Aktivierungsfunktion spielt bei der Abbildung des Wissens eine bedeutende Rolle. Beim Einsatz linearer Funktionen zur Aktivierung der Neuronen können Probleme bei der Darstellung nichtlinearen Wissens auftreten. KNN sind nur insofern eine gute Wahl, wenn sie vor ihrem Einsatz gut konzipiert und trainiert worden sind.

2.3.3 Lernregeln

Je nach Aufgabenfeld und angestrebtem Ziel ist es wichtig, das KNN richtig anzulernen. Dazu können verschiedene Ansätze verfolgt werden. Die am häufigsten angewendete Methode stellt das Adaptieren der im Netz vorhandenen Gewichte dar. Weitere, weniger häufig angewendete Varianten des Lernens in KNN sind:

- die Adaption der Schwellwerte der Aktivierungsfunktion der Neuronen
- Modifikation (Hinzufügen/Entfernen) von Verbindungen zwischen den Neuronen
- Veränderung der Propagierungs-/Aktivierungs-/Ausgabefunktion
- Modifikation der Anzahl der Neuronen.

(Vgl. [5], S.37)

Im Folgenden wird das „Lernen“ in einem KNN immer mit dem Verändern der Gewichte des Netzes gleichgesetzt. Zu Beginn aller Lernverfahren werden alle Gewichte zufällig, mit einem nicht zu großen Startwert initialisiert. Zum Anlernen der Gewichte werden bekannte Datensätze verwendet, wobei der größte Teil davon zum Trainieren, der kleinere zum Validieren der Ergebnisse genutzt wird. Die Trainingsdatensätze können dabei einzeln präsentiert und somit das Netz angelernt werden. Man spricht dann vom inkrementellen Lernen. Eine weitere Möglichkeit zur Gewichtsadaption bietet das epochale Lernen. Hierbei werden dem Netz zunächst alle Trainingsdatensätze präsentiert und anschließend die Gewichte verändert.

2.3.3.1 Hebb'sches Lernen

Die Grundidee des Hebb'schen Lernens besteht darin, dass genau dann eine Gewichtsveränderung zwischen zwei Neuronen realisiert wird, wenn zwischen diesen beiden eine Aktivität besteht. Sind also beide zur gleichen Zeit aktiv, wird das Gewicht zwischen ihnen verändert.

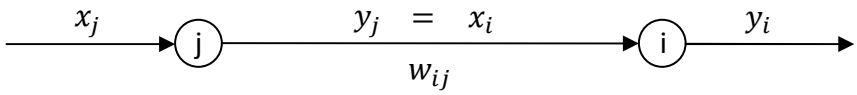


Abbildung 4: Gewichtsänderung $\Delta w_{ij} \neq 0 \leftrightarrow y_{i,j} = 1$

Folgende Gleichungen beschreiben diesen Zusammenhang:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \text{ und} \tag{2.17}$$

$$\Delta w_{ij} = \varepsilon \cdot y_i \cdot y_j, \tag{2.18}$$

wobei Δw_{ij} die Gewichtsänderung, ε den Lernparameter ($0 < \varepsilon \leq 1$), y_i die Ausgabe (bzw. den Aktivitätslevel) des empfangenden Neurons und y_j die des sendenden Neurons beschreibt. Wird $\varepsilon = 1$ gewählt, tritt der triviale Fall $\Delta w_{ij} = y_i \cdot y_j$ ein, die Gewichtsänderung entspricht genau dem Produkt aus den Ausgaben der beiden aktiven

Neuronen. Der Lernabbruch kann z. B. durch die Einführung einer Schranke s realisiert werden, wobei das Lernen beendet wird, wenn keine Gewichtsänderungen in allen Gewichten mehr vorliegen ($\Delta w_{ij} < s, \forall i, j$). Auch eine maximale Anzahl an Lerndurchläufen könnte als Abbruchkriterium genutzt werden. Einsatz findet diese Lernregel in allen Lernmethoden. Die hier vorgestellten Formeln realisieren ein inkrementelles Lernen, doch durch geringfügige Anpassung ist auch ein epochales Lernen möglich (vgl. [5], S.38ff.).

2.3.3.2 Delta-Lernalgorithmus

Dieser relativ einfache Lernalgorithmus ist eine Form des überwachten Lernens. Dabei werden die tatsächliche Ausgabe ($y_{i,ist}$) und die erwartete Ausgabe ($y_{i,soll}$) eines Neurons i miteinander verglichen. Ihre Differenz bildet den Delta-Wert ($\delta_i = y_{i,soll} - y_{i,ist}$) des Neurons. Stimmt die erwartete Ausgabe nicht mit der berechneten überein, müssen die Gewichte des betreffenden Neurons folgendermaßen korrigiert werden:

- $\delta > 0$: Gewichte mit positiver Verbindung müssen erhöht werden.
  Gewichte mit negativer Verbindung müssen reduziert werden.

- $\delta = 0$: Es muss keine Veränderung der Gewichte vorgenommen werden.

- $\delta < 0$: Gewichte mit positiver Verbindung müssen reduziert werden.
  Gewichte mit negativer Verbindung müssen erhöht werden.

Die neuen Gewichte berechnen sich wie folgt:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \text{ und} \tag{2.19}$$

$$\Delta w_{ij} = \varepsilon \cdot \delta_i \cdot y_j. \tag{2.20}$$

Der Wert ε stellt den Lernparameter dar, y_j ist die Ausgabe des sendenden Neurons j. Mit Hilfe dieses Schemas werden alle das Neuron betreffenden Gewichte in Abhängigkeit zum Lernparameter und zur Ausgabe des jeweiligen vorgeschalteten Neurons adaptiert. Starke Inputs y_j in das Neuron i gewährleisten durch ihre multiplikative Verknüpfung mit dem Delta-Wert eine große Gewichtsänderung (vgl. [5], S.40ff.).

Bei Netzen mit einer oder mehreren Hidden-Schichten ist diese Lernregel nicht anwendbar, da nur die direkte Ausgabe eines Neurons beobachtet und verglichen werden kann. Weitere Bezeichnungen für die Delta-Regel sind Widrow-Hoff-Regel oder Least Mean Square Regel (LMS, kleinstes mittleres Quadrat) (vgl. [3], S.113).

2.3.3.3 Gradientenabstiegsverfahren

Das Gradientenabstiegsverfahren ist ein mathematischer Algorithmus zur Minimierung des Gesamtfehlers der Ausgabe. Der nach einer Anregung des Netzes beobachtete Fehler E ist abhängig von den im Netz vorhandenen Gewichten, also $E = f(w)$. Verändert man die Gewichte, verändert sich der Gesamtfehler (bei gleichem Ein-/Ausgabe Muster). Grafisch lässt sich diese Abhängigkeit als Gütegebirge in einem n-dimensionalen Raum darstellen ($n-1$ Gewichte). Abbildung 5 zeigt ein zweidimensionales Gütegebirge. Eine gebräuchliche Definition des Fehlerterms ist:

$$E = \sum_{1}^{m} (t_i - y_i)^2, \tag{2.21}$$

wobei y_i die beobachtete und t_i die gewünschte Ausgabe darstellen und m der Anzahl der Ausgangsneuronen des Netzes entspricht. Der Fehler entspricht demnach den aufsummierten Quadraten der aus der beobachteten und gewünschten Beobachtungen entstehenden Differenzen. Die Aufgabe des Algorithmus besteht darin, diesen Fehler sukzessive zu verringern und den kleinsten Gesamtfehler E_{min} zu finden. Um dies zu erreichen, wird zunächst ein zufälliger Startpunkt festgelegt und der Gradient dieses Punktes berechnet. Aufgrund der Gradientenbildung ist die Verwendung von differenzierbaren Propagierungs- und Aktivierungsfunktionen beim Anwenden dieses Verfahrens unerlässlich. Um die neuen Gewichte zu berechnen, wird der Lernschritt in Richtung des kleinsten Gradienten im Gütegebirge ausgeführt.

$$w^{neu} = w^{alt} + \Delta w \tag{2.22}$$

Im zweidimensionalen Fall:

$$\Delta w = -\varepsilon \frac{\partial f(w^{alt})}{\partial w} \tag{2.23}$$

Im n-dimensionalen Fall:

$$\Delta w = -\varepsilon \cdot \nabla f(w^{alt}), \text{ mit} \tag{2.24}$$

$$\nabla = \sum_{1}^{n} \frac{\partial}{\partial \mathrm{x_i}} \cdot e_{\mathrm{x_i}}. \tag{2.25}$$

Die Schrittweite ε gibt dabei an, wie weit im Gütegebirge vorangeschritten wird. Der Vorgang wird abgebrochen, wenn kaum noch Änderungen in den Gewichten auftreten oder eine maximale Anzahl von Iterationen durchgeführt wurde.

Beim Anwenden dieses Algorithmus können viele Probleme auftreten, die, um Falschinterpretationen der Ergebnisse zu vermeiden, beseitigt werden müssen. Wird z. B. die Schrittweite unzureichend angepasst oder falsch initiiert, könnte der Fall auftreten, dass globale und auch lokale Minima nicht mehr gefunden werden können. Durch zu große Schrittweiten können Oszillationen zwischen Gebirgswänden hervorgerufen werden. Das Abbruchkriterium „Brich bei kleiner Gewichtsänderung ab!“ kann zu unzureichenden Ergebnissen führen, da bei Sattelpunkten oder bei sehr flachen Plateaus auch ein Abbruch des Algorithmus geschieht. Beseitigt werden können diese Probleme z. B. durch die Wahl verschiedener Startpunkte oder eine adaptive Schrittweitenanpassung (Vergrößerung bei flachen Plateaus; Verkleinerung bei Oszillation) und den Einsatz eines sogenannten Momentum-Terms. Dieser beachtet nicht nur den aktuellen Gradienten, sondern auch den mit einer Konstante gewichteten vorherigen Gradienten. Mathematisch kann dies folgendermaßen dargestellt werden:

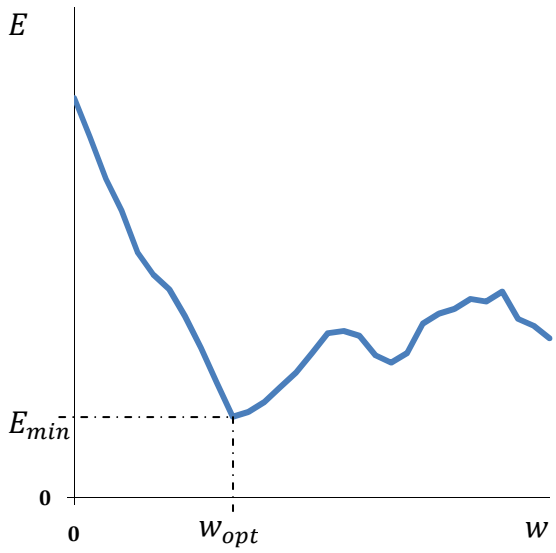


Abbildung 5: 2-dimensionales Gütegebirge (ein Gewicht)

$$\nabla f(.)_{\mathrm{gesamt}} = \nabla f(.)_{\mathrm{aktuell}} + \beta \cdot \nabla f(.)_{\mathrm{alt}} \tag{2.26}$$

Dadurch werden lokale Minima eher vermieden, flache Plateaus schneller durchlaufen und auch das Problem der Oszillation kann dadurch relativiert werden. Bei ungünstigen Gebirgslandschaften kann durch den Einsatz dieser Methode allerdings auch das globale Minimum durchlaufen werden, ohne dass ein Abbruchkriterium greift. Eine weitere Möglichkeit bietet die Delta-Bar-Regel, bei der nicht nur ein zurückliegender Gradient in die Berechnung mit einbezogen wird, sondern alle zurückliegenden Gradienten. Dabei werden die weit in der Vergangenheit liegenden Werte schwächer gewichtet als weniger weit zurückliegende Werte (vgl. [5], S.42ff.).

Auch das Gradientenabstiegsverfahren kann aufgrund des nur direkt zu korrigierenden Fehlers nur bei Netzen ohne Hidden-Schicht angewendet werden.

2.3.3.4 Backpropagation-Algorithmus

Die bisher vorgestellten Verfahren bieten keine Möglichkeit zum Anlernen von Netzen mit einer oder mit mehreren Hidden-Schichten. Das Problem ist offensichtlich: der bei einem präsentierten Beispieldatensatz beobachtete Fehler zwischen gewollter und tatsächlicher Ausgabe kann nur direkt am Ausgang des Netzes beobachtet werden, da an dieser Stelle der Soll- /Ist-Vergleich vorgenommen wird. Auf diese Weise können die Gewichte zwischen Hidden- und Ausgangsschicht korrigiert werden. Wie es jedoch zu dem beobachteten Fehler in den vorgeschalteten Schichten kam, bleibt unklar. Bei Netzen mit Hidden-Schicht(en) tragen auch die gewichteten Verbindungen dieser zum entstanden Fehler bei. Der Backpropagation-Algorithmus versucht, den am Ausgang entstandenen Fehler auf die Gewichte im Netz, die Anteil an seiner Entstehung hatten, zurück zu koppeln und diese dadurch zu korrigieren, oder: „Bei der Backpropagation werden die Gewichtungen der verborgenen Schichten anhand der Fehler aus der folgenden Schicht angepasst [sic!].“ ([3], S.162).

Grundlage für die Adaption ist dabei das Gradientenabstiegsverfahren. Zunächst wird die Änderung der Gewichte der Output-Schicht mit Hilfe folgender Gleichungen berechnet:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \tag{2.27}$$

$$\Delta w_{ij}^{(l)} = \varepsilon \cdot y_j^{(l-1)} \cdot \delta_i^{(l)} \tag{2.28}$$

$$\delta_i^{(l)} = \left(t_i^{(l)} - y_i^{(l)}\right) \cdot g', \tag{2.29}$$

wobei g' die Ableitung der Aktivierungsfunktion des Neurons i der Output-Schicht darstellt. Die Änderung der Gewichte ist also zum einen abhängig vom Lernparameter ε, der zugehörigen Aktivierung $y_j^{(l-1)}$ ($= x_i^{(l)}$) und zum anderen vom Wert $\delta_i^{(l)}$.

Die Änderung der Gewichte der Vorgänger-Schicht (Schicht $l-1$) berechnet sich wie folgt:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \tag{2.30}$$

$$\Delta w_{ij}^{(l-1)} = \varepsilon \cdot y_j^{(l-2)} \cdot \delta_i^{(l-1)} \tag{2.31}$$

$$\delta_i^{(l-1)} = \left(\sum_{k=1}^{m} \delta_k^{(l)} \cdot w_{ik}^{(l)}\right) \cdot g', \tag{2.32}$$

Der Unterschied zur Berechnung in der Output-Schicht (Schicht l) besteht darin, dass nicht mehr der direkte Fehler am Ausgang $(t_i - y_i)$ zur Berechnung des Delta-Terms und somit der neuen Gewichte genutzt wird, sondern die aufsummierten Produkte aus den Delta-Werten $\delta_k^{(l)}$ und den korrespondierenden Gewichten $w_{ik}^{(l)}$. Auch dieses Verfahren kann durch einen Momentum-Term erweitert werden.

Ein weiteres Problem, dass bei der Anwendung dieses Verfahrens auftreten kann, ist die Verlangsamung der Konvergenz des Lernens beim Lernen an den Grenzbereichen der Neuronen und der Verwendung sigmoidaler Aktivierungsfunktionen. Werden viele Beispieldatensätze verwendet, welche die Neuronen nur in den Grenzbereichen ihrer Aktivierungsfunktionen ansprechen, verlangsamt sich das Lernen drastisch. Dieses Problem liegt in der Ableitung der Sigmoid-Funktion begründet. Diese hat ihr Maximum an der Stelle $x = 0$ und fällt zu beiden Seiten logarithmisch ab (vgl. Gauss-Funktion). Durch die multiplikative Verknüpfung des Ableitungs-Terms (siehe (2.29) und (2.32)) mit den restlichen Faktoren zur Berechnung des δ-Wertes entstehen nur kleine Änderungen bei der Verwendung von sehr großen bzw. kleinen x-Werten der Aktivierungsfunktion. Dieses Problem kann durch die Addition einer Konstante zur Ableitung positiv beeinflusst werden, da hierdurch eine Multiplikation mit einem minimalen Wert verschieden von Null gewährleistet wird.

2.3.3.5 Competitive Learning

Dieses Verfahren adaptiert die Gewichte im KNN abhängig davon, welches Output-Neuron bei der Präsentation einer Eingabe die größte Reaktion aufweist. Dabei wird dem Netz zunächst ein Beispiel-Muster präsentiert, alle Ausgänge der Output-Schicht berechnet und der „Gewinner" bestimmt. Die Gewichte, die am „Siegerneuron" anliegen werden verstärkt, alle anderen Gewichte bleiben unverändert. Auf diese Weise findet eine Art Kategorisierung statt. Die neuen Gewichte werden folgendermaßen berechnet:

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \tag{2.33}$$

$$\Delta w_{ij} = \varepsilon \cdot (y_j - w_{ij}). \tag{2.34}$$

Die Lernrate ε entscheidet darüber, wie stark die Gewichte geändert werden. Die Differenz $(y_j - w_{ij})$ entscheidet darüber, ob die Gewichte erhöht ($(y_j - w_{ij}) > 0$), gesenkt ($(y_j - w_{ij}) < 0$) oder nicht verändert ($(y_j - w_{ij}) = 0$) werden.

2.3.4 Lernmethoden künstlicher neuronaler Netze

Eine Lernregel beschreibt den Algorithmus zum Erlernen des Wissens. Die Lernmethode hingegen beschreibt, wie das abzubildende Wissen erlernt wird. Für die Anwendung von KNN gibt es sehr viele Möglichkeiten und die zu beschreibenden Informationen liegen nicht immer in expliziter Form vor. Im einfachsten Fall verfügt der Anwender jedoch über ausreichend a priori-Informationen, um das KNN direkt anzulernen. In der Trainingsphase werden dem Netz E-/A- Muster präsentiert, aus welchen das Netz sein Wissen erzeugt. In diesem Fall spricht man vom **überwachten Lernen**, da nach der Präsentation eines Trainingsbeispiels die vom Netz erzeugte und die gewünschte (bekannte) Ausgabe verglichen werden. Aus dem resultierenden Fehler werden Rückschlüsse zur Adaption des Netzes gezogen. Diese Methode findet sich z. B. bei der Hebb'schen Lernregel, der Delta-Regel, dem Backpropagation Verfahren und einigen stochastischen Verfahren.

Beim **nicht überwachten Lernen** überwacht kein Lehrer den Lernprozess und das Netz passt seine Ausgabe an Gleichmäßigkeiten im Trainingsdatenraum an. Es werden Merkmale von

Objekten erlernt, welche später eine Zuweisung zu verschiedenen Klassen ermöglichen (vgl. [3], S.42f). Dem KNN werden also nur Eingangsdaten präsentiert. Anwendung findet diese Methode vor allem dann, wenn Klassifizierungen vorgenommen werden sollen oder bei der Mustererkennung. Klassische Vertreter, bei denen diese Lernmethode zur Anwendung kommt, sind Kohonen- und Hamming-Netze. Außerdem sind das Competitive Learning und das ART-Modell in diesem Zusammenhang zu erwähnen.

Sind die korrekten Antworten auf die möglichen Eingangsdaten in das Netz nicht bekannt, sondern nur ein Ziel, dass es zu erreichen gilt, spricht man vom **bestärkenden Lernen**. Die Adaption des Netzes geschieht durch Ausprobieren, d. h. die durch eine oder mehrere Aktionen erhaltene **Belohnung** wird verwendet, um das gestellte Ziel bestmöglich zu erreichen (am schnellsten, am energiesparendsten, am kürzesten etc.). Lern-Automaten bilden einen Anwendungsfall.

Bei den **direct design methods** findet keine Gewichtsveränderung statt, die Struktur des Netzes wird zu Beginn beschlossen. Diese Methode sei nur am Rande erwähnt, da bei ihrer Anwendung kein Lernen im Sinne der Gewichtsadaption stattfindet (vgl. [5], S.30).

Kapitel 3
Regelung des inversen Pendels

Die vorgestellten Konzepte des Reinforcement Learning und der künstlichen neuronalen Netze bieten viele Anwendungsmöglichkeiten. Im Folgenden sollen sie miteinander verknüpft und dazu verwendet werden, eine stabile Regelung für das inverse Pendel zu ermöglichen. Dazu wird zunächst der allgemeine Aufbau beleuchtet. Anschließend werden die einzelnen Komponenten erklärt und es wird aufgezeigt, wie diese realisiert wurden.

3.1 Aufbau und Struktur der Regelung

Im Vorfeld der Regelung am realen Pendel wurde ein „Anlernen“ des Reinforcement Learning System (RLS) durch den Einsatz eines hinreichend genauen Modells in MATLAB/Simulink durchgeführt, da der Zeitaufwand zum Anlernen am realen System im Allgemeinen zu groß ist. Dazu wurde das durch theoretische Modellbildung erstellte, lineare Modell aus einer vorangegangenen Studienjahresarbeit ([8]) verwendet. Es wurde erweitert, um einen Neustart des Systems nach dem „Umkippen“ des Pendels zu ermöglichen (Fehlerfall angenommen bei Winkel des Pendels $\varphi \geq \pm 10°$ oder Position des Wagens $x_w \geq \pm 0{,}4m$ (Begrenzung der Schiene erreicht)). Dieser Reset ist notwendig, da zum Anlernen des RLS ein sehr häufiges Eintreten des Fehlerfalls notwendig ist. Das auf diese Weise angelernte System kann am realen Pendel eingesetzt und durch eine geeignete Programmierung für die jeweilige Anwendung (Minimierung/Entfernung des „Slipstick“-Effekts, Minimierung der Pendelauslenkung, Minimierung des Zeithorizontes bis die Regelung abgeschlossen ist etc.) adaptiert werden. Ein Anlernen am realen Pendel würde ein Rücksetzen des Pendels im Fehlerfall erzwingen. Dies wäre jedoch bei einer hohen Anzahl von Lernschritten mit einem enormen zeitlichen und manuellen Aufwand verbunden.

Ein RLS lernt, wenn der Agent durch Aktionen auf seine Umwelt einwirkt, um diese dahingehend zu beeinflussen, sein ihm vorgegebenes Ziel zu erfüllen. Dies löst ein von der Umwelt zurückgekoppeltes Signal (Reinforcement, Reward) aus, welches auf die Güte der

getätigten Aktion schließen lässt und zum Adaptieren der vom Agenten verwendeten Taktik genutzt wird (siehe Kapitel 2.1). Das Pendelmodell und die seine Dynamiken beschreibenden Differentialgleichungen definieren die Umwelt. Die vier beobachteten Zustände Position des Wagens x_w (=x_1), Geschwindigkeit des Wagens $\dot{x}_w$ (=x_2), Winkel des Pendels φ (=x_3) und Winkelgeschwindigkeit des Pendels $\dot{\varphi}$ (=x_4) werden als Eingänge in den Agenten und zur Generierung eines Fehlersignals genutzt. Die verwendeten Werte für das Fehlersignal sind 0 (kein Fehler aufgetreten) und -1 (das Pendel ist umgekippt oder der Wagen ist in die Begrenzung gefahren). Dieses Fehlersignal fungiert als weiterer Eingang in den Agenten. Die verfolgte Taktik wird durch die Aktionsfunktion (AF) beschrieben, welche den aktuellen Zuständen eine Aktion zuordnet. Weiterhin ist die Wertefunktion (WF) zu realisieren, welche den Zuständen einen Wert zuordnet. Diese beiden Funktionen werden durch je ein KNN abgebildet. Ihre Gewichte repräsentieren das im RLS gespeicherte Wissen und müssen im Laufe des Lernprozesses angepasst werden. Diese Realisierung entspricht den in Kapitel 2.2.6. vorgestellten Actor-Critic-Methoden, da eine explizite Trennung von Taktik und Wertegenerierung vorliegt. In Abbildung 6 ist der allgemeine Aufbau dieser Realisierung zu erkennen (vgl. [2], S.151ff.).

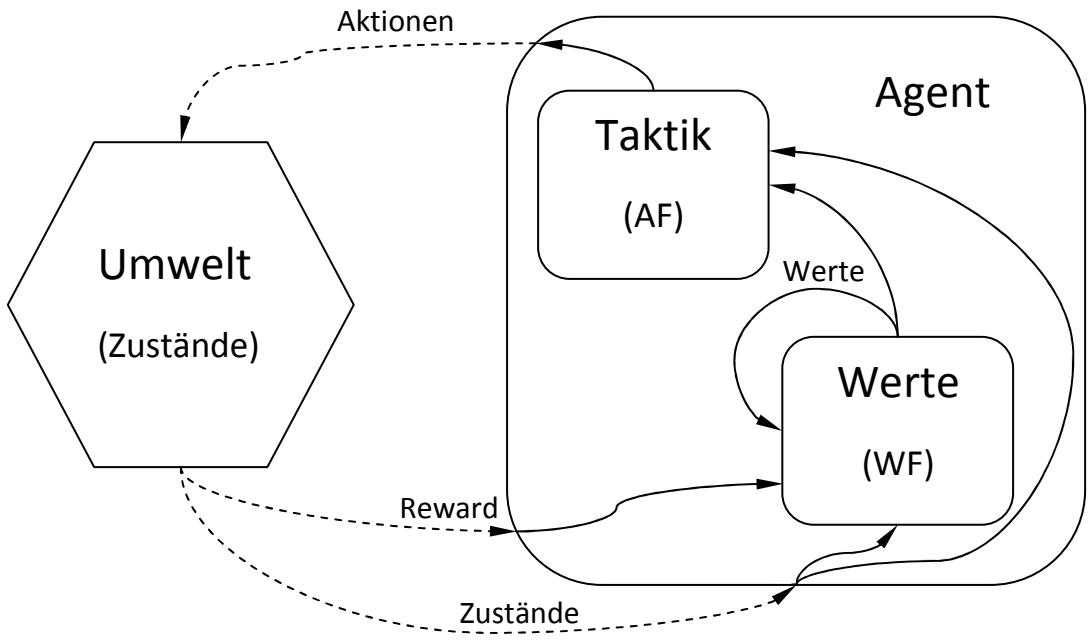


Abbildung 6: Allgemeiner Aufbau eines RLS mit Actor-Critic-Methode

3.2 Komponenten und deren Realisierung

Im Folgenden wird die Realisierung des RLS in MATLAB/Simulink aufgezeigt und erläutert. Abbildung 7 zeigt das Simulink-Modell *„RLS_KNN_Simulation.mdl“*. Es bildet die Basis der erstellten Lösungen und enthält alle Komponenten des RLS.

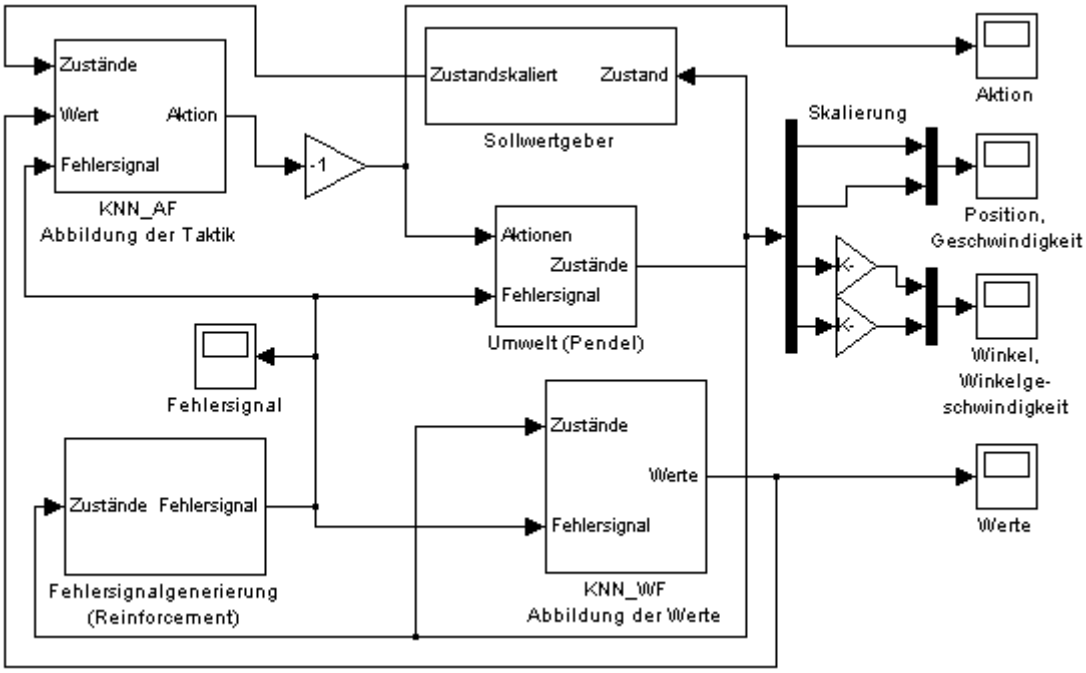


Abbildung 7: Simulink-Modell *„RLS_KNN_Simulation.mdl“*

3.2.1 Das Umweltmodell

Die Zustandsdifferentialgleichungen des Pendelmodells wurden mit Hilfe von Blöcken der Standard-Simulink-Bibliothek nachgebildet und dahingehend erweitert, dass die Startzustände beim Eintreten des Fehlerfalles wieder angenommen werden können. Dazu müssen die im System enthaltenen Integratoren zurückgesetzt werden. Abbildung 8 zeigt diese Erweiterungen auf.

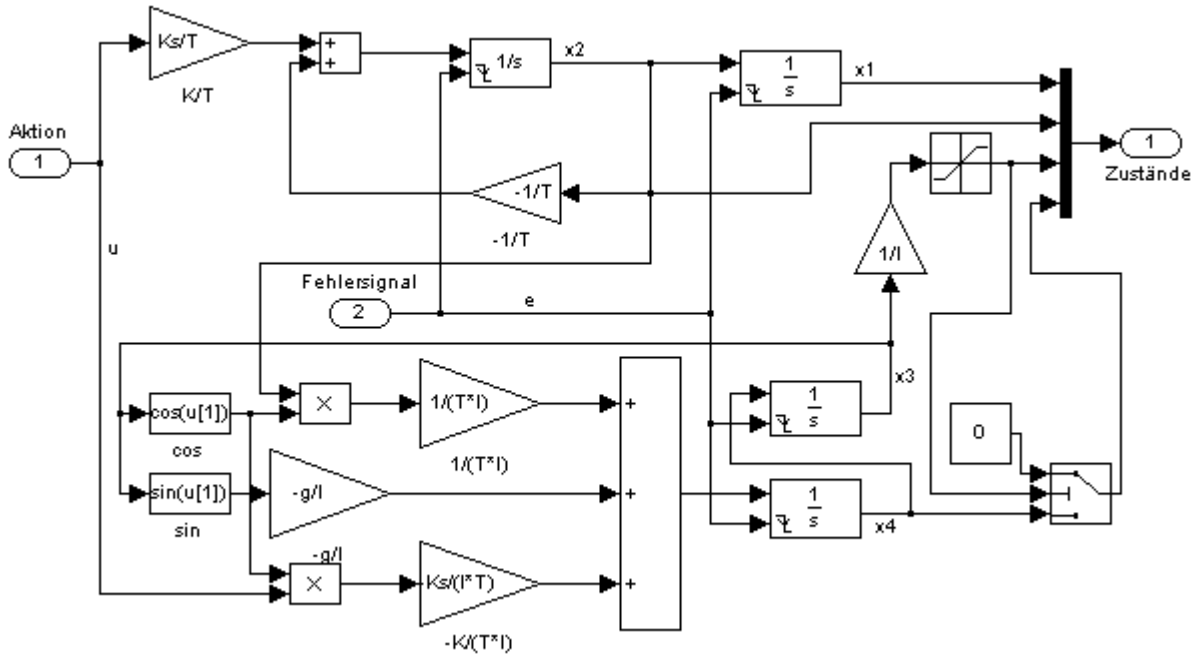


Abbildung 8: Simulink-Modell des Pendels; Reset durch Fehlersignal

Als Eingang in dieses System dienen zum einen die Stellgröße u (Aktion) und zum anderen das Fehlersignal e; seine Ausgänge entsprechen den vier Zustandsvariablen des Pendels. Abbildung 9 zeigt dieses die Umwelt des RLS beschreibende Subsystem (SuSy).

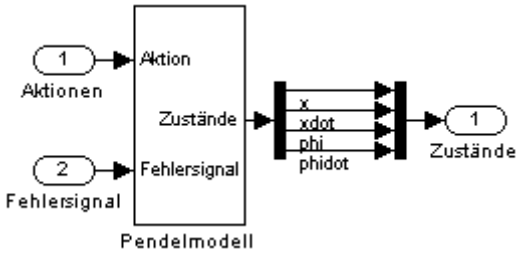


Abbildung 9: Subsystem des Pendels

3.2.2 Bildung des Fehlersignals

Die Bildung des Fehlersignals wird mit einem weiteren SuSy realisiert. Das System wird in Abbildung 7 mit „Fehlersignalgenerierung (Reinforcement)“ bezeichnet. Die Zustände Position des Wagens (x_1) und Winkel des Pendels (x_3) werden als Eingänge in das System genutzt und daraus mit Hilfe von Standard-Blöcken aus der Simulink-Bibliothek das Fehlersignal generiert, welches den Ausgang des System darstellt. Wird dieses Signal mit einem Wert von -1 belegt, ist der Fehlerfall eingetreten, das Pendel ist also entweder umgekippt oder der Wagen ist in die linke oder rechte Begrenzung gefahren. Tritt kein Fehler auf, wird das Signal mit dem Wert 0 belegt. Das Fehlersignal stellt das Reinforcement-Signal

dar, also die Rückmeldung aus der Umwelt für den Agenten, und bildet die Grundlage zum Lernen im System.

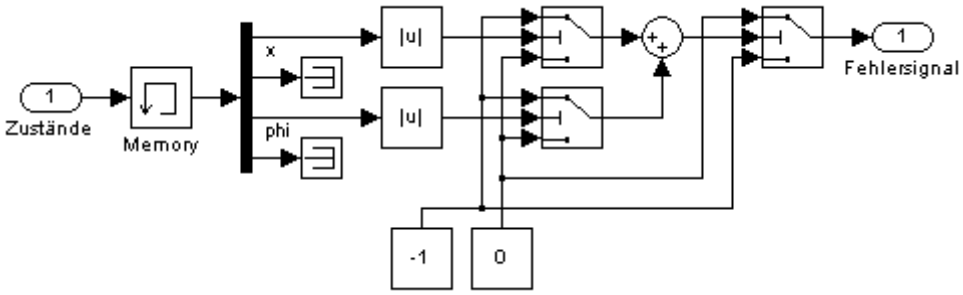


Abbildung 10: Subsystem zur Fehlergenerierung

3.2.3 Der Agent

Die Kernkomponente des RLS verbirgt sich in den zwei letzten Subsystemen. Diese repräsentieren den Agenten und damit verbunden die Abbildung der Taktik und der Wertefunktion. Sie bestehen aus jeweils einem KNN, welche die Aktionsfunktion und die Wertefunktion approximieren. Beiden Netzen dienen die Zustände der Umwelt und das Fehlersignal als Eingänge.

3.2.3.1 KNN_WF: Abbildung der Wertefunktion

In dem die Wertefunktion abbildenden KNN (im Folgenden mit KNN_WF bezeichnet) werden den aktuellen Zuständen Werte zugeordnet, die eine Aussage darüber treffen, wie gut oder schlecht es ist, sich im jeweiligen Zustand zu befinden. Dies ist notwendig, da ein Lernen, dass nur auf dem von der Umwelt zugeordneten Reinforcement (-1 im Fehlerfall, 0 in allen anderen Zuständen) basiert, sehr langsam ist. Das KNN_WF wird durch Abbildung 11 illustriert. Befindet sich das Pendel in Ruhelage ($x_3 \approx 0$ und $x_4 \approx 0$), wird diesem Zustand ein großer Wert $V(z_t) \to 0$ zugeordnet werden, da das Auftreten eines Fehlers (das Umkippen des Pendels) eher unwahrscheinlich ist. Im umgekehrten Fall, bei $x_3 \to \pm 12°$ und $|x_4| \gg 0$, ist das Auftreten eines Fehlers sehr wahrscheinlich und der Wert dieser Zustandskonstellation wird bei $V(z_t) \to -1$ liegen. Ein weiterer Fehlerfall kann eintreten,

sobald der Wagen die Beschränkungen der Schiene erreicht, also $|x_1| \geq 0{,}4m$. Demnach werden Zustandskonstellationen mit $|x_1|$ in der Nähe von $0{,}4m$ und entsprechender Wagengeschwindigkeit x_2 ($x_1 \to +0{,}4m$ und x_2 positiv bzw. $x_1 \to -0{,}4m$ und x_2 negativ) mit kleinen Werten $V(z_t) \to -1$ korrespondieren.

Die vom KNN_WF berechneten Werte dienen dem Agenten in der Zeit, in der das Fehlersignal den Wert 0 einnimmt, als „Ersatz-Reinforcement“. Er nutzt die Änderung der Werte (also die Änderung des Ausgangs des KNN_WF), um die Effektivität der vorrangegangenen Aktion zu bewerten. Eine positive Wertänderung korrespondiert demnach mit einem Übergang in einen Zustand, in dem das Auftreten eines Fehlers weniger wahrscheinlich ist und umgekehrt. Diese Werte werden durch die im Kapitel 2.2.4 gezeigten TD-Verfahren erlernt.

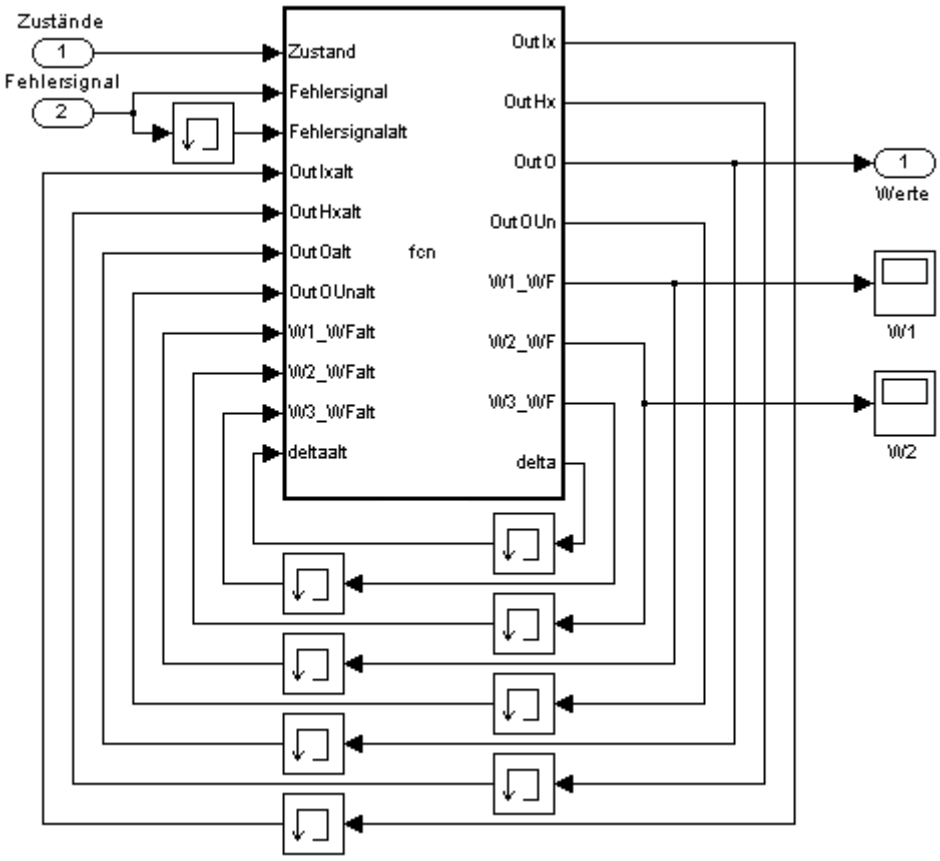


Abbildung 11: KNN_WF

3.2.3.2 KNN_AF: Abbildung der Aktionsfunktion

Zusätzlich zu den Zuständen und dem Fehlersignal dient die zeitliche Änderung des Ausganges aus dem KNN_WF als weiterer Eingang in das zweite KNN des Agenten. Dieses bildet die Aktionsfunktion ab (im Folgenden wird dieses KNN mit „KNN_AF“ bezeichnet) und stellt als Ausgang die Aktion des Agenten bereit. Im Fall des Pendels ist dieser Wert die auf den Wagen wirkende Kraft (bzw. die zur Kraftaufbringung benötigte Motorspannung). Dieses Netz führt die Abbildung des jetzigen Zustandsvektors $x(t)$ auf die korrespondierende Aktion $a(t)$ durch. Sie stellt folglich das Stellglied des Agenten dar, denjenigen Teil, mit dem er aktiv auf die Umwelt einwirken kann. Der Agent hat die globale Aufgabe, das in jedem Zeitschritt erhaltene Reinforcement über den gesamten Zeithorizont zu maximieren. Er wird demzufolge diejenigen Aktionen bevorzugen, welche ein großes Reinforcement (bzw. einen hohen Wert aus dem KNN_WF) zur Folge haben. Durch Abbildung 12 wird die Realisierung des KNN_AF in Simulink näher erläutert.

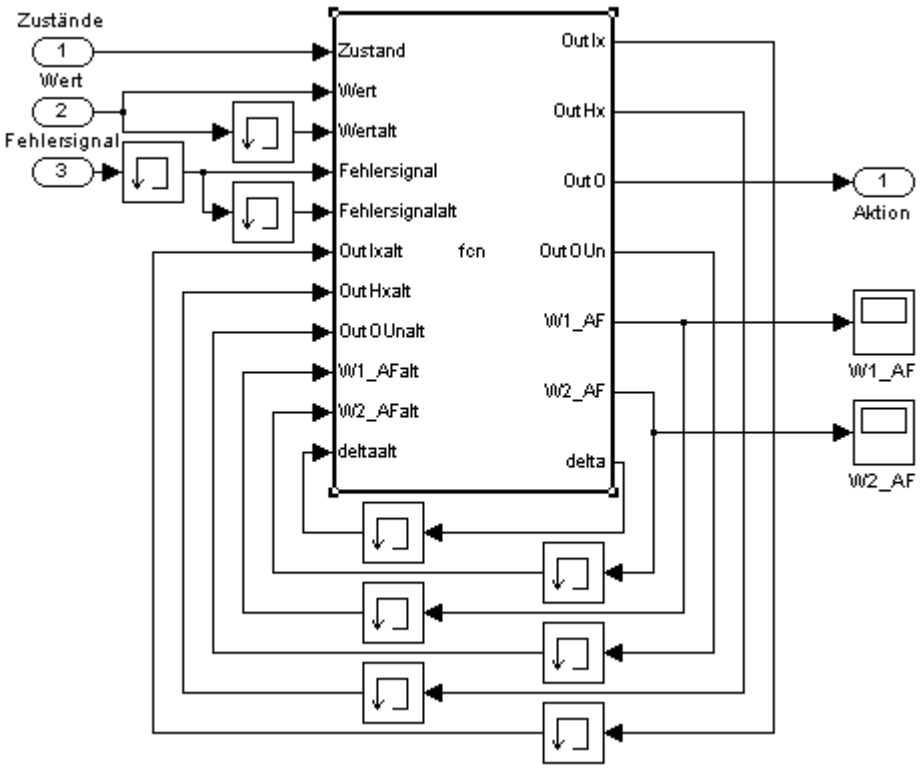


Abbildung 12: KNN_AF

3.2.3.3 Dimensionierung der Netze

Wie in Kapitel 2.3.2 erwähnt, können nichtlineare Funktionen nur mit Netzen abgebildet werden, welche eine (oder mehrere) Hiddenschicht(en) besitzen. Dies muss bei der Dimensionierung der KNN_AF und KNN_WF beachtet werden. Die Nichtlinearität der Wertefunktion WF kann relativ einfach verdeutlicht werden, indem man sich vor Augen führt, wann ein Fehler eintritt und welche Wertzuordnung dies für Zustände nahe einer Fehlerzustandskonstellation zur Folge hat. Das in Abbildung 13 gezeigte Diagramm fasst dies für die Zustände x_3 und x_4 zusammen. Es handelt sich dabei um eine qualitative Darstellung. Analog dazu kann eine Untersuchung mit Bezug auf die Zustände x_1 und x_2 durchgeführt werden.

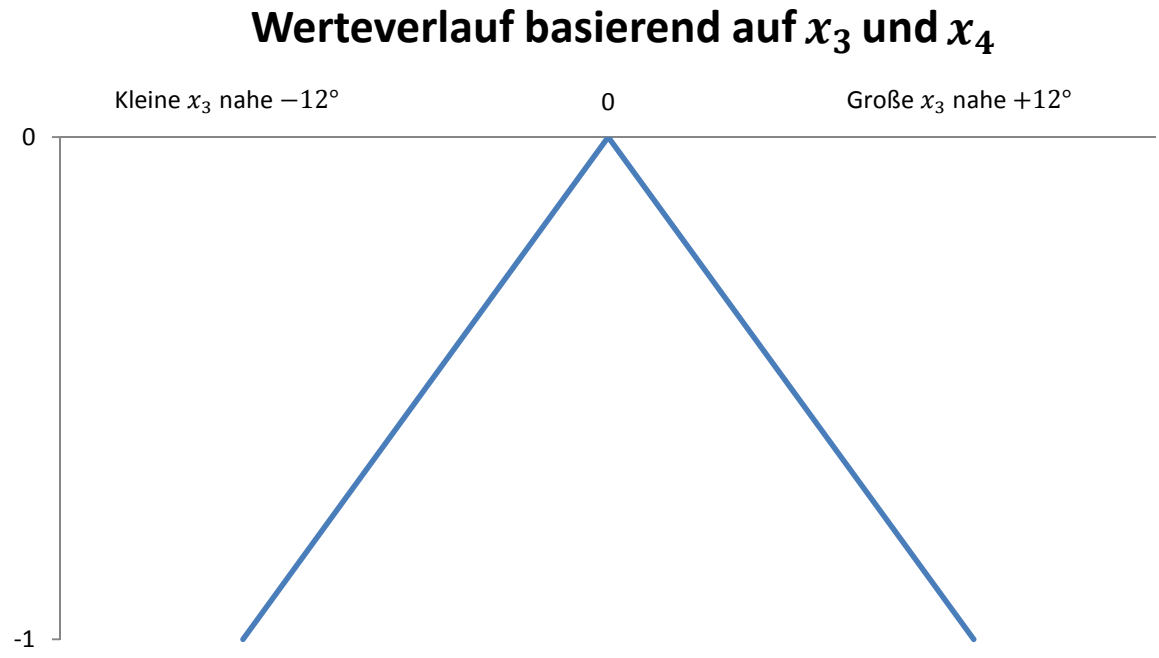


Abbildung 13: Qualitativer Werteverlauf bezüglich der Zustände x_3 und x_4

Diese Funktion kann nicht mit einem einschichtigen KNN abgebildet werden. Dies muss bei der Erstellung des KNN_WF beachtet werden. Einen Ausweg aus dieser Problematik bietet die Möglichkeit, den Zustandsraum in viele kleine Regionen zu unterteilen und diesen Regionen je einen binären Eingang in das KNN_WF zuzuordnen (vgl. [6], S.33). Zu jedem

Zeitpunkt wäre so gewährleistet, dass nur eine Region aktiv ist und das Netz beansprucht und für jede Zustandskonstellation (beliebig genau durch Diskretisierung) genau ein Gewicht existiert und angelernt wird. Diese Realisierung würde die Verwendung eines einschichtigen Netzes erlauben.

Eine weitere Möglichkeit zur Abbildung dieser Nichtlinearität bietet die Verwendung einer Hiddenschicht im KNN_WF. Im Rahmen dieser Arbeit wurde diese Möglichkeit wahrgenommen und realisiert.

Es existiert ein lineares Regelgesetz zum Lösen dieser Aufgabe. Das KNN_AF könnte demnach auf Basis eines einschichtigen Netzes realisiert werden. Da jedoch die Nichtlinearitäten in der Strecke im aufgestellten Modell nicht beachtet wurden, wird auch im Falle dieses Netzes ein mehrschichtiges Netz verwendet. Damit können im Betrieb am realen Pendel die u. a. durch Reibung auftretenden (nichtlinearen) Effekte u. U. berücksichtigt und die nötigen Änderungen in den Gewichten adaptiert werden.

Die Anzahl der Neuronen in den Schichten hängt im Falle der Eingangsschicht von den jeweiligen Eingängen in das Netz ab. Das KNN_WF besitzt vier Input-Neuronen, für jeden Zustand je ein Neuron. Das Fehlersignal wird als indirekter Eingang in dieses Netz genutzt. Das Signal wird nicht auf ein Eingangsneuron aufgeschaltet, es wird zum Anlernen der Gewichte als Reinforcement verwendet. Auch das KNN_AF besitzt vier Neuronen in dieser Schicht und die Zustände dienen auch hier als Eingänge. Die Anzahl der Neuronen der Hiddenschicht wurde experimentell ermittelt und bei beiden Netzen mit fünf festgelegt. In der Ausgangsschicht des Netzes ist jeweils nur ein Neuron implementiert, welches den jeweiligen Ausgang des Netzes berechnet (KNN_WF → Wert des aktuellen Zustandes; KNN_AF → Aktion des Agenten).

Jedes Neuron (mit Ausnahme der Inputneuronen) besitzt desweiteren eine gewichtete Verbindung zu einem Bias-Neuron. Die verwendeten Netze sind vollvermascht und nicht rekurrent (vgl. Kapitel 2.3.1). Das KNN_WF besitzt weiterhin Verbindungen von den Eingangsneuronen direkt zum Ausgangsneuron. Es zeigte sich, dass das Lernverhalten in diesem Netz auf diese Weise verbessert (schnelleres Erlernen der hinreichend korrekten Werte) werden kann.

3.2.3.4 Struktur der Neuronen

Signalskalierung

Bevor die Zustandssignale auf die Eingangsneuronen aufgeschaltet werden können, müssen sie skaliert werden. Der Grund dafür liegt in den in den Neuronen verwendeten Aktivierungsfunktionen. Ihr Ausgabebereich ist beschränkt, bei der Verwendung der Tangens-Hyperbolicus-Funktion (siehe Kapitel 2.3.1) liegt dieser Bereich beispielsweise bei $-1 \leq y \leq 1$. Schaltet man jedoch einen Zustand, z. B. die Position des Pendelwagens, auf ein Neuron das diese Aktivierungsfunktion verwendet, unskaliert auf, wird man deren Wertebereich schnell ausreizen. Der Definitionsbereich der Funktion ist zwar mit $-\infty \leq x \leq \infty$ definiert, jedoch verändern sich die Funktionswerte bei großen Werten für x nur geringfügig. Eine Skalierung sollte also in der Form:

$$\hat{x} = \frac{x - c}{d - c} \cdot (b - a) + a \tag{3.1}$$

erfolgen. Der normierte Eingang in das KNN $\hat{x}$ wird aus den Definitionsbereichsgrenzen des KNN a und b und den Definitionsbereichsgrenzen c und d der Eingangsvariable x ermittelt, wobei gilt: $x \in \{c; d\}$ und $\hat{x} \in \{a; b\}$. Das vorliegende Problem erlaubt eine Vereinfachung, da beide Wertebereiche den Mittelwert 0 besitzen. Außerdem wird durch die Annahme $\hat{x} \in \{-1; 1\}$ eine weitere Vereinfachung möglich. Die Berechnung des normierten Eingangs kann also mit Hilfe der Formel

$$\hat{x} = \frac{x}{d} \tag{3.2}$$

durchgeführt werden. Einem Maximalwert von $x_3 = 10°$ wird demnach beispielsweise ein Input in das Neuron von 1 zugeordnet.

Das gleiche Problem gilt für das Ausgabeneuron. Der Pendelwagen kann mit einer Spannung von $\pm 10V$ angesteuert werden. Die verwendeten Ausgabeneuronen sowie alle anderen verwendeten Neuronen arbeiten mit der Tangens-Hyperbolicus-Funktion als Aktivierungsfunktion. Wie bereits oben erwähnt, besitzt diese einen Wertebereich von $y \in \{-1{,}1\}$. Beachtet man diese Problematik nicht, geschieht Folgendes: der unskalierte Ausgang verhindert ein Erreichen des gestellten Regelungszieles und der Lernalgorithmus versucht weiterhin die Gewichte zu verändern. Ein Konvergieren dieser ist somit sehr

unwahrscheinlich. Nimmt man an, dass $y \in \{a, b\}$ den Wertebereich des skalierten Ausgangssignals und $\hat{y} \in \{c, d\}$ den Wertebereich des Output-Neurons des KNN darstellen, beseitigt in Analogie zur obigen Formel die Anwendung folgender Berechnungsvorschrift zur Skalierung dieses Problem:

$$y = \frac{\hat{y} - c}{d - c} \cdot (b - a) + a. \tag{3.3}$$

Im Falle des KNN_AF ist eine Vereinfachung aufgrund der Mittelwerte der Wertebereiche der verwendeten Aktivierungsfunktion und der Motorspannung möglich.

$$y_{KNN\_AF} = \hat{y}_{KNN\_AF} \cdot b. \tag{3.4}$$

Das KNN_WF hingegen besitzt einen Ausgangswertebereich von $-1 \leq y_{KNN\_WF} \leq 0$. Die Skalierung lässt sich auf die Formel:

$$y_{KNN\_WF} = 0.5 \cdot (\hat{y}_{KNN\_WF} - 1) \tag{3.5}$$

reduzieren.

Propagierungsfunktion

Die in allen Neuronen (Input-Neuronen ausgenommen, diese besitzen keine gewichtete Verbindung) verwendete Propagierungsfunktion verknüpft die jeweiligen Gewichte multiplikativ mit den Eingangssignalen. Der Input x_i berechnet sich folgendermaßen:

$$x_i^{(l)} = \sum_{j=1}^{n} y_j^{(l-1)} \cdot w_{ij}\,, \tag{3.6}$$

wobei n die Anzahl der Neuronen der vorhergehenden Schicht $(l-1)$ angibt, $y_j$für den Output des Neurons j der vorhergehenden Schicht $(l-1)$ steht und w_{ij} das jeweilige Gewicht zwischen den Neuronen darstellt.

Aktivierungsfunktion

Wie bereits erwähnt, wird in allen Neuronen die Tangens-Hyperbolicus Funktion als Aktivierungsfunktion verwendet. Die Entscheidung hierfür fiel zum einen aufgrund der einfachen Skalierung der Signale und zum anderen, weil die Funktion differenzierbar ist. Dies erlaubt die Verwendung des Backpropagation-Algorithmus zum Anlernen der Gewichte (vgl. Kapitel 2.3.3.4).

3.3 Lernverfahren

Um zu verstehen, wie ein intelligentes System fast ohne a priori-Wissen dazu in der Lage ist, nur mit Hilfe eines relativ seltenen Reinforcements seitens der Umwelt eine Regelung zu bewerkstelligen, muss man sich erneut die Algorithmen des Reinforcement Learning vor Augen führen. Der Agent versucht aufgrund der von der Umwelt erhaltenen Rückmeldung Schlüsse auf seine getätigten Aktionen zu ziehen und somit die ihm gestellte Aufgabe zu lösen. Dazu wird er zunächst lernen die Situationen (Zustände), in denen er sich befindet, einzuschätzen, ihnen also Werte zuzuordnen. Sobald dies hinreichend geschehen ist, kann er folgende Fragen beantworten: „Wie gut ist es, dass ich mich in diesem Zustand befinde?“ und durch die Beobachtung zweier aufeinanderfolgender Zustände: „Wie gut war meine getätigte Aktion?“. Die Beantwortung der letzten Frage ermöglicht dem Agenten nun die Aufstellung einer geeigneten Taktik, um sein ihm vorgegebenes Ziel zu erreichen.

Dieses Vorgehen sieht ein sequentielles Lernen von Wertefunktion und Aktionsfunktion (Taktik) vor. Ein paralleles Lernen von Taktik und Zustandswerten ist ebenso denkbar, jedoch sollten die Ergebnisse, die zu Beginn des Lernvorgangs erzielt wurden auch unter Vorbehalt behandelt werden. Die ungefilterte Interpretation der zu diesem Zeitpunkt ermittelten, höchstwahrscheinlich inkorrekten Zustandswerte kann zur Ausführung einer im Bezug auf das gestellte Ziel unvorteilhaften Taktik führen. Desweiteren kann der Lernprozess dadurch verlangsamt werden.

Ein weiterer Punkt der hier Beachtung finden sollte, ist die Exploration/Exploitation (vgl. Kapitel 2.2.4). Der vorgegebene Zustandsraum sollte möglichst vollständig erforscht werden, um eine optimale Problemlösung zu garantieren. RLS, die keine vollständige Exploration vornehmen, neigen dazu, ihr Lernen einzustellen, sobald sie eine Lösung für das ihnen gegebene Problem gefunden haben. Kapitel 4.1.3 fasst diese Problematik erneut auf und erläutert sie genauer.

3.3.1 Anlernen der Wertefunktion

Bevor der Lernvorgang gestartet werden kann, müssen einige Initialwerte festgelegt werden. Dazu gehören zum einen die Startzustände des Pendels, welche auch nach jedem Fehlerauftritt wieder angenommen werden und zum anderen die Startwerte der Gewichte in den Netzen. Bei der Wahl Letzterer ist zu beachten, dass die Lerngeschwindigkeit und auch generell die Konvergenz des Verfahrens von ihnen abhängen können. Im Falle der Wertefunktion haben sich Anfangsgewichte mit Werten zwischen -1 und $+1$ bewährt. Außerdem müssen die Lernparameter α, γ und ε des Verfahrens festgelegt werden, worauf im Verlauf dieses Kapitels noch genauer eingegangen wird.

Nachdem die Werte initiiert wurden, kann der Lernvorgang für die Wertefunktion gestartet werden. Um eine möglichst vollständige Erforschung des Zustandsraumes zu erreichen, sollten die vom Agenten getätigten Aktionen relativ zufällig verteilt sein. Allerdings sei an dieser Stelle erwähnt, dass auch diese Zufälligkeit aufgrund der Dynamik des Systems zunächst keine vollständige Erforschung des Zustandsraumes erlaubt. Dies liegt vor allem daran, dass das Umkippen des Pendels viel schneller erfolgen kann als das Fahren des Wagens in die Begrenzung, es sei denn, man initiiert die Wagenposition relativ nah an der Begrenzung der Schiene. Um eine Zufälligkeit in der Wahl der Aktionen des Agenten zu erreichen, empfiehlt es sich zu Beginn des Lernens nicht den Ausgang des noch ungelernten KNN_AF als Aktionsgeber zu verwenden, sondern einen unabhängigen Zufallsgenerator. Dieser wählt im Bereich $u_{min} \leq u \leq u_{max}$ pro Zeitschritt einen Wert aus, welcher dem Pendelsystem als Stellsignal aufgeschaltet wird. Würde man das ungelernte KNN_AF zur Aktionsgenerierung verwenden, so würde die gewählte Stellamplitude zu Beginn sehr stark von den gewählten Anfangsgewichten des Netzes abhängen. Eine zufällige Aktionswahl ist somit nicht unbedingt gewährleistet. Möchte man jedoch von Beginn an nicht auf die Verwendung des KNN_AF verzichten, bietet folgende leichte Adaption des Netzes eine Abhilfe für dieses Problem. Der Ausgang des KNN_AF wird mit einem Zufallssignal mit Mittelwert 0 verrauscht. Somit wird sichergestellt, dass die Startwerte der Gewichte dieses Netzes relativ klein sind und damit auch der vom Netz generierte Ausgang. Auf diese Weise kann zunächst eine relativ zufällige Aktionsgenerierung erfolgen. Zusätzlich sollte der Lernparameter des Netzes sehr klein gewählt werden, damit die Adaption der Gewichte nur sehr langsam voran schreitet. Nach einer ausreichend langen Lernphase (und Erforschung

des Zustandsraumes) könnte dieser anschließend erhöht werden und somit die Gewichte so stark adaptiert werden, dass das auf den Eingang aufgeschaltete Rauschen nur noch einen minimalen Einfluss hat. Dazu sollte beachtet werden, dass das Rauschen nicht zu groß gewählt wird. Alternativ könnte man den Rauschterm auch entfernen, sobald das Anlernen des KNN_WF hinreichend genau angelernt wurde.

Wie in Kapitel 3.2.3.3 erwähnt, wurden die Netze mit einer Hiddenschicht versehen. Dies hat zur Folge, dass das Anlernen der Gewichte im Netz mit Hilfe des Backpropagation-Algorithmus ausgeführt werden muss. Beschrieben ist dieser zunächst für den Fall des Supervised Learning, ein Lernverfahren, dass das Vorhandensein von Beispieldatensätzen voraussetzt. Die Formeln zur Adaption der Gewichte lauten demnach (vgl. Kapitel 2.3.3.4):

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \quad \text{(Output- und Hidden-Schicht)} \quad (3.7)$$

$$\Delta w_{ij}^{(l)} = \varepsilon \cdot y_j^{(l-1)} \cdot \delta_i^{(l)} \quad \text{(Output-Schicht)} \quad (3.8)$$

$$\delta_i^{(l)} = \left(t_i^{(l)} - y_i^{(l)}\right) \cdot g' \quad \text{(Output-Schicht)} \quad (3.9)$$

$$\Delta w_{ij}^{(l-1)} = \varepsilon \cdot y_j^{(l-2)} \cdot \delta_i^{(l-1)} \quad \text{(Hidden-Schicht)} \quad (3.10)$$

$$\delta_i^{(l-1)} = \left(\sum_{k=1}^{m} \delta_k^{(l)} \cdot w_{ik}^{(l)}\right) \cdot g' \quad \text{(Hidden-Schicht)} \quad (3.11)$$

Da in diesem Fall der Beispieldatensatz ($t_i^{(l)}$ – teach-Wert) fehlt, muss eine Möglichkeit gefunden werden, um ihn dem Netz zu präsentieren. Er stellt den Sollausgang des Netzes dar, also im Fall des KNN_WF denjenigen Wert, der dem aktuellen Zustand zugeordnet werden soll. Diese Werte sind jedoch zu Beginn der Lernphase unbekannt und müssen erlernt werden. Man steht also vor dem Problem des Lernens im Lernen: das Netz muss angepasst werden an eine Sollausgabe, die wiederum auch erst erlernt werden muss. Wie dies geschehen kann, zeigen uns die in Kapitel 2.2.4 erklärten Algorithmen des TD-Lernens. Zunächst wird angenommen, dass die Summe aus aktuellem Wert eines Zustandes $V(z_t)$ und dem aktuell erhaltenen Reward r_t eine neue, bessere Schätzung, das Update des alten Wertes $V(z_{t-1})$ darstellt. Die Änderung des alten Zustandes wird gemäß folgender Formel vorgenommen:

$$V^{neu}(z_{t-1}) = V^{alt}(z_{t-1}) + \alpha \cdot \left(r_t + \gamma \cdot V(z_t) - V^{alt}(z_{t-1})\right). \quad (3.12)$$

Es wird also eine mit α diskontierte Differenz aus altem und neuen Wert und neuem erhaltenen Reward zum alten Wert hinzuaddiert. Auf die künstlichen neuronalen Netze angewendet bedeutet das, dass der neue teach-Wert sich aus dieser Formel errechnet. D. h.:

$$t_i^{(l)} = V^{neu}(z_{t-1}). \tag{3.13}$$

Da die Ausgabe des KNN_WF relativ (abhängig von den Startwerten der Gewichte) zufällig ist, werden zunächst nur kleine Änderungen der Werte bei verschiedenen Zuständen erfolgen. Sobald der Fehlerfall eingetreten ist, ist die erste große Änderung im teach-Wert zu beobachten. In diesem Fall ist der erhaltene Reward $r_t = -1$ und $V(z_t) = 0$ und ein Update des Zustandswertes kurz vor dem Fehlerfall wird vorgenommen. Im nächsten Fehlerfall wird eine erneute Korrektur durchgeführt. Da die Zustände, die kurz vor dem Fehler auftreten, bereits mit einem kleineren Wert adaptiert wurden, hat dieser Zustandswert einen Einfluss auf seinen Vorgänger und auch die Zustände, die zwei Zeitschritte vor dem Fehlerfall beobachtet werden, werden adaptiert. Legt man den Zeitpunkt des Fehlers mit t fest, lässt sich nun schon eine Änderung der Zustandswerte bei $t - 2$ festestellen.

Das Verfahren konvergiert zu den Werten $-1 \leq y_{WF} \leq 0$. Die Geschwindigkeit mit der es konvergiert hängt dabei sehr stark von den gewählten Parametern α, γ und ε ab. Im Laufe der Untersuchung haben sich die Parameterkombinationen $\alpha = 0{,}1, \gamma = 0.95$ und $\varepsilon = 0.05$ als praktikabel für das KNN_WF erwiesen. Es sei an dieser Stelle darauf hingewiesen, dass kleinere α zwar ein genaueres Ergebnis in der Abbildung der Wertefunktion zur Folge haben, die zur Generierung der ausreichend korrekten Zustandswerte jedoch einen größer werdenden zeitlichen Aufwand bedeuten. Das Verhältnis zwischen der erreichten Genauigkeit und der aufgewendeten Zeit rechtfertigt die Verkleinerung der α nur bis zu einem bestimmten Punkt, der von der Leistung des verwendeten Rechners und der Problemstellung abhängt.

Es sollte weiterhin darauf geachtet werden, dass das Lernen nach einem Fehlerfall für den Zeitpunkt nach dem Auftreten des Fehlers ausgesetzt wird. Wird dies nicht beachtet, werden die Werte der Zustände zum Zeitpunkt des Resets (also die zu Beginn festgelegten Startwerte) und die Werte zum Zeitpunkt des Auftretens des Fehlers miteinander verglichen, obwohl der Übergang dieser beiden Zustände kein natürlich durch das Pendel hervorgerufener, sondern ein durch das Reset erzwungener Vorgang ist.

Ein Lernschritt für das KNN_WF hat folgenden Ablauf:

1. Vorwärtspropagierung des Inputsignals durch das Netz; Berechnung $\hat{y}_{WF,t}$ (Netzausgang) und $y_{WF,t}$ (normierter Netzausgang). Speicherung der Ausgänge aus Input-, Hidden- und Output-Schicht und normierter Ausgang.
2. Vergleich aktuell berechneter Zustandswert $y_{WF,t}$ mit gespeichertem Wert $y_{WF,t-1}$ und daraus Generierung des teach-Wertes nach folgenden Formeln:

   - Im Fehlerfall:

   $$y_{WF,t-1}^{neu} = y_{WF,t-1}^{alt} + \alpha \cdot \left(-1 + \gamma \cdot 0 - y_{WF,t-1}^{alt}\right) \quad (3.14a)$$

   - Tritt kein Fehler auf:

   $$y_{WF,t-1}^{neu} = y_{WF,t-1}^{alt} + \alpha \cdot \left(0 + \gamma \cdot y_{WF,t} - y_{WF,t-1}^{alt}\right) \quad (3.14b)$$

   Berechnung des neuen $teach$-Wertes:

   $$teach_{t-1} = y_{WF,t-1}^{neu} \quad (3.15)$$

3. Präsentation des errechneten teach-Wertes $teach_{t-1}$ als Grundlage für den Backpropagation-Algorithmus. Berechnung der neuen Gewichte nach den Formeln:

   - Für die Output-Schicht gilt:

   $\delta_i^{(2)} = \left(teach_{t-1} - y_{WF,t-1}^{alt}\right) \cdot \left(\hat{y}_{WF,t-1} + 1\right) \cdot \left(1 - \hat{y}_{WF,t-1}\right)$, mit (3.16a)
   $\left(\hat{y}_{WF,t-1} + 1\right) \cdot \left(1 - \hat{y}_{WF,t-1}\right)$: Abl. Aktivierungsfkt. Output-Unit

   $\Delta w_{ij}^{(2)} = \varepsilon \cdot \hat{y}_j^{(1)} \cdot \delta_i^{(2)}$, mit $\hat{y}_j^{(1)}$: Output der Hidden-Unit j (3.17a)

   - Für die Hidden-Schicht gilt:

   $\delta_i^{(1)} = \left(\sum_{k=1}^{m} \delta_k^{(2)} \cdot w_{ik}^{(2)}\right) \cdot \left(\hat{y}_{WF,H_i,t-1}^{(1)} + 1\right) \cdot \left(1 - \hat{y}_{WF,H_i,t-1}^{(1)}\right)$, mit (3.16b)
   $\left(\hat{y}_{WF,H_i,t-1}^{(1)} + 1\right) \cdot \left(1 - \hat{y}_{WF,H_i,t-1}^{(1)}\right)$: Abl. Aktivierungsfkt. Hidden-Unit i
   $\Delta w_{ij}^{(1)} = \varepsilon \cdot \hat{y}_j^{(0)} \cdot \delta_i^{(1)}$, mit $\hat{y}_j^{(0)}$: Output der Input-Unit j (3.17b)

   - Die neuen Gewichte berechnen sich für alle Schichten folgendermaßen:

   $$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \quad (3.18)$$

Eine Aktualisierung erfolgt also immer für den vorhergehenden Wert. Deshalb müssen die alten Aktivitäten des Netzes auch zwischengespeichert werden. Im Simulink-Modell wurde dies durch die Memory Blöcke realisiert (vgl. Abb. 11).

Da das Lernen online durchgeführt wird, kann die Konvergenz des Verfahrens am zeitlichen Verlauf des Ausganges des Netzes überprüft werden. Wie bereits erwähnt, konvergieren die Werte zunächst für große $|x_3|$ nahe der Fehlergrenze zum Wert -1. Ist dieser Vorgang hinreichend genau erfolgt, kann mit dem Anlernen der AF begonnen werden bzw. der oben erwähnte Rauschterm am Ausgang des KNN_AF entfernt werden. Es sei hier noch einmal ausdrücklich darauf hingewiesen, dass die Wertefunktion noch nicht ausgelernt ist, sie liefert zu diesem Zeitpunkt aufgrund des oben beschriebenen Problems nur für die Zustände x_3 und x_4, also den Winkel und die Winkelgeschwindigkeit der Auslenkung des Pendels, hinreichend korrekte Werte. Sobald der Zustandsraum durch eine entsprechende Aktionswahl einer frühen Version der AF hinsichtlich der Zustände x_1 und x_2 besser erforscht ist, kann die WF angepasst werden und liefert hinreichend korrekte Werte für alle Zustände.

3.3.2 Anlernen der Aktionsfunktion

Der Agent kann mit Hilfe einer hinreichend bekannten Wertefunktion eine Aussage darüber treffen, wie gut seine Entscheidung in den jeweiligen Zuständen war. Er benötigt zum Anlernen der Aktionsfunktion die Differenz zwischen zwei aufeinanderfolgenden Zustandswerten um diese Einschätzung durchführen zu können. Zunächst jedoch müssen einige Initialwerte gesetzt werden, um mit dem Lernprozess beginnen zu können. Da auch hier ein KNN angelernt wird, ist zunächst die Lernrate ε zu wählen.

Das KNN_AF beobachtet die Zustände des Pendels und lernt diejenigen Entscheidungen zu treffen, welche einen Anstieg in den Zustandswerten zur Folge haben. Wie im vorherigen Kapitel beschrieben, ist zu Beginn des Lernprozesses der Aktionsfunktion die Wertefunktion nur insofern bekannt, als dass sie die hinreichend korrekten Werte für die Zustände x_3 und x_4 ausgibt. Daraus resultiert zwangsläufig, dass zunächst das Netz dahingehend adaptiert wird, Aktionen zu erlernen, die den durch das Umkippen des Pendels hervorgerufenen

Fehlerfall zu vermeiden versuchen. Zu erkennen ist dies zum einen in einem Anstieg der Zeit zwischen zwei durch Umkippen verursachte Fehler und zum anderen in einer größer werdenden Amplitude des Stellgliedes. Die Gewichte des Netzes werden also zunächst dahingehend adaptiert, dass der Ausgang des KNN_AF das Pendel stabilisiert, ohne jedoch auf die Grenzen des Wertebereiches des Zustandes x_1 zu achten. Demzufolge fährt der Wagen nun in die linke bzw. rechte Beschränkung. Zu diesem Zeitpunkt ist ein Weiterlernen des KNN_AF nur bedingt möglich, da die zum Lernen benötigten Informationen aus dem KNN_WF noch nicht vorliegen. Ein Lernen des KNN_WF ist nun jedoch möglich, da der Zustandsraum bezüglich der Variablen x_1 und x_2 nun durch das Fahren des Wagens in die seitlichen Begrenzungen erforscht werden kann.

Ob das Lernen im KNN_AF zu diesem Zeitpunkt ausgesetzt und das Erlernen der korrekten Werte durch das KNN_WF abgewartet wird, oder ob ein paralleles Lernen durchgeführt wird, bleibt dem Anwender überlassen. Im Rahmen dieser Arbeit wurden beide Ansätze untersucht und beide erwiesen sich als praktikabel.

Sobald die Wertefunktion auch die Werte für die Position und die Geschwindigkeit des Wagens erlernt hat, kann das KNN_AF dahingehend adaptiert werden, dass es durch geeignete Aktionen die Beschränkungen des Zustandes x_1 nicht mehr verletzt. Im Fall des KNN_AF kommen die gleichen Formeln zur Berechnung der neuen Gewichte wie beim KNN_WF zum Einsatz. Der einzige Unterschied zum Lernverfahren beim die Wertefunktion erlernenden Netz besteht in der Bestimmung des Delta-Termes. In diesem Fall wird eine Adaption darüber vorgenommen, wie sich die Werte $y_{WF,t-1}$ und $y_{WF,t}$ zweier aufeinanderfolgender Zustände verändert haben. Die Differenz

$$TD = y_{WF,t} - y_{WF,t-1} \tag{3.19}$$

wird zur Bestimmung des Delta-Terms eingesetzt und gibt damit die Veränderung der Gewichte vor. Ist diese Differenz positiv, hat eine Zustandsüberführung in einen Zustand mit einem größeren Wert stattgefunden, das System befindet sich also in einem besseren Zustand bezüglich des formulierten Regelungsziels. In Zukunft sollte diese Aktion bei gleichen Zuständen also bevorzugt bzw. noch verstärkt werden. Ist die Differenz negativ, so ist das Gegenteil der Fall, die Aktion hat zu einem schlechteren Zustand geführt und sollte in

dieser Form nicht mehr gewählt werden. Diese Differenz wird als temporale Differenz (TD) bezeichnet und stellt das einzige benötigte Signal zum Erlernen der AF und auch der WF dar.

Ein Lernschritt für das KNN_AF hat also folgenden Ablauf:

1. Vorwärtspropagierung des Inputsignals durch das Netz; Berechnung $\hat{y}_{AF,t}$ (Netzausgang) und $y_{AF,t}$ (normierter Netzausgang). Speicherung der Ausgänge aus Input-, Hidden- und Output-Schicht.
2. Nutzen der TD zur Evaluierung der im vorherigen Zeitschritt gespeicherten Aktion. Belohnung/Bestrafung der Aktion mit Hilfe des Terms: $TD \cdot \hat{y}_{AF,t-1}$.
3. Adaption der Gewichte nach der Formel:

   - Für die Output-Schicht gilt:

   $$\delta_i^{(2)} = TD \cdot \hat{y}_{AF,t-1} \cdot \left(\hat{y}_{AF,t-1} + 1\right) \cdot \left(1 - \hat{y}_{AF,t-1}\right), \text{ mit} \quad (3.20a)$$
   $\left(\hat{y}_{AF,t-1} + 1\right) \cdot \left(1 - \hat{y}_{AF,t-1}\right)$: Abl. Aktivierungsfkt. Output-Unit

   $$\Delta w_{ij}^{(2)} = \varepsilon \cdot \hat{y}_j^{(1)} \cdot \delta_i^{(2)}, \text{ mit } \hat{y}_j^{(1)} : \text{ Output der Hidden-Unit } j \quad (3.21a)$$

   - Für die Hidden-Schicht gilt:

   $$\delta_i^{(1)} = \left(\sum_{k=1}^{m} \delta_k^{(2)} \cdot w_{ik}^{(2)}\right) \cdot \left(\hat{y}_{AF,H_i,t-1}^{(1)} + 1\right) \cdot \left(1 - \hat{y}_{AF,H_i,t-1}^{(1)}\right), \text{ mit} \quad (3.20b)$$
   $\left(\hat{y}_{AF,H_i,t-1}^{(1)} + 1\right) \cdot \left(1 - \hat{y}_{AF,H_i,t-1}^{(1)}\right)$: Abl. Aktivierungsfkt. Hidden-Unit i

   $$\Delta w_{ij}^{(1)} = \varepsilon \cdot \hat{y}_j^{(0)} \cdot \delta_i^{(1)}, \text{ mit } \hat{y}_j^{(0)} : \text{ Output der Input-Unit } j \quad (3.21b)$$

   - Die neuen Gewichte berechnen sich für alle Schichten folgendermaßen:

   $$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \quad (3.22)$$

3.4 Die Sollposition

Die Hauptaufgabe der Regelung ist die Stabilisierung des Pendels. Ein weiteres Ziel ist die Überführung des Wagens in eine ihm vorgegebene Sollposition. Bei dieser Sprungantwort des gesamten Systems darf das Pendel nicht umkippen, was beim Anlernen der Aktionsfunktion beachtet werden muss. Um diese Überführung zu realisieren, wird eine

Skalierung des aktuellen Zustandes x_1^{ist} vorgenommen, welche von der vorgegebenen Sollposition (x_1^{Soll}) und von der verwendeten Methode abhängt. Dieser Wert x_1^{skal} dient dem KNN_AF als neuer Eingang des Zustandes x_1. Wird ein Wert $x_1^{Soll} \neq 0$ gewählt, versucht das Netz die Ruhelage wieder einzunehmen und den Sollzustand zu erreichen. Es sind verschiedene Formen der Neuskalierung denkbar.

Eine lineare Verschiebung wird demnach wie folgt realisiert:

$$x_1^{skal} = x_1^{ist} - x_1^{Soll}. \tag{3.23}$$

Die Anwendung dieser einfachen Skalierung hat jedoch den Nachteil, dass durch die Verschiebung der Geraden die Maximal- und Minimalwerte nicht mehr eingehalten werden (siehe Abb.14).

Eine nichtlineare Verschiebung kann mit Hilfe folgender Formel erzielt werden:

$$x_1^{skal} = \begin{cases} -\frac{x_1^{min}}{x_1^{Soll}-x_1^{min}} \cdot x_1^{ist} + \frac{x_1^{Soll} \cdot x_1^{min}}{x_1^{Soll}-x_1^{min}}; x_1^{ist} < x_1^{Soll} & ; \quad x_1^{min} < x_1^{ist} < x_1^{Soll} \\ 0 & ; \quad x_1^{ist} = x_1^{Soll} \\ \frac{x_1^{max}}{x_1^{max}-x_1^{Soll}} \cdot x_1^{ist} - \frac{x_1^{max} \cdot x_1^{Soll}}{x_1^{max}-x_1^{Soll}}; x_1^{ist} > x_1^{Soll} & ; \quad x_1^{max} > x_1^{ist} > x_1^{Soll} \end{cases}. \tag{3.24}$$

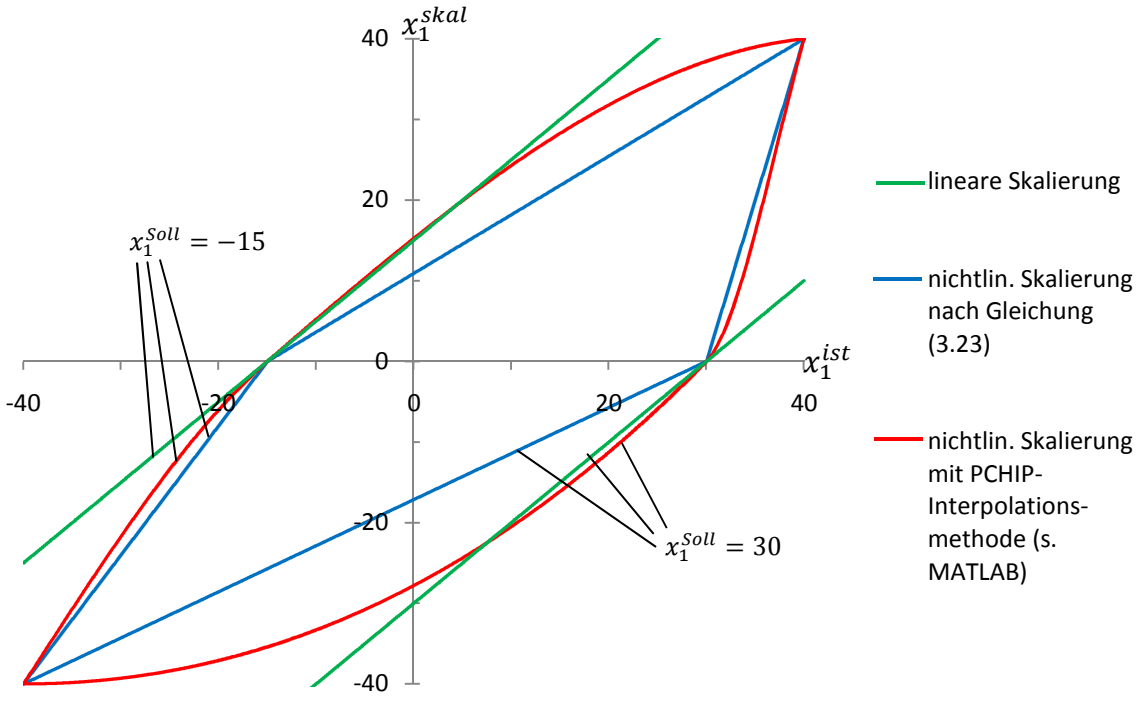


Abbildung 14: Darstellung verschiedener Skalierungen unter Verwendung verschiedener Interpolationsverfahren

Durch diese Skalierung entsteht eine veränderte Abbildung des Zustandes x_1 in Abhängigkeit von x_1^{Soll} und der jeweils verwendeten Interpolationsmethode. Diese funktionale Abhängigkeit wird in Abbildung 14 anhand verschiedener Sollpositionen erklärt.

3.5 Einsatz am realen Pendel

3.5.1 Versuchsaufbau

Der Versuchsaufbau besteht aus drei Hauptelementen. Zum einen der Pendelaufbau selbst. Das Pendel ist auf einem Wagen befestigt, der auf einer zu beiden Seiten begrenzten Schiene läuft. Zum Anderen die Rapid Control Prototyping (RCP) Umgebung der Firma dSPACE, auf der die Regelalgorithmen abgelegt werden. Außerdem ist diese für die Signalaufarbeitung und -weiterleitung zum dritten Element verantwortlich, einem PC mit Schnittstelle und der erforderlichen Software. Dieses Kapitel beschäftigt sich zunächst mit der mechanischen Beschreibung des Pendels um danach auf die Signalweiterleitung und die Elemente dSPACE-Box und Anwender-PC genauer einzugehen.

3.5.1.1 Das Pendel

Der Wagen und das auf ihm stehende Pendel befinden sich in einem Gestell, das zum universelleren Einsatz so konzipiert wurde, dass es auch zur Simulation eines Krans, also mit hängendem Pendel, eingesetzt werden kann. Der Wagen befindet sich auf einer Schiene, die bei einer Gesamtlänge von $80cm$ zu beiden Seiten durch Anschläge begrenzt ist. Er ist an einem Endlosriemen befestigt, welcher über zwei Umlenkrollen läuft. Eine dieser Umlenkrollen ist mit einem Gleichstrommotor gekoppelt und fungiert somit als Antrieb für den Wagen. Das auf dem Wagen stehende Pendel hat eine Länge von ca. $73cm$ und kann am freien Ende durch Gewichte ergänzt werden, wodurch sich verschiedene Systemdynamiken generieren lassen. Die Bewegung des Pendels ist so begrenzt, dass es nur einen Freiheitsgrad besitzt und nur entlang des Fahrtweges des Wagens pendeln kann. Auf diese Weise wird

sichergestellt, dass der Wagen das Pendel balancieren kann. Um die Zustände des Pendels und des Wagens erfassen zu können, wurden drei Sensoren installiert. Die Position des Wagens wird über einen rotatorischen Inkrementalgeber mit linearer Kennlinie ermittelt. Durch den direkt mit der Welle des antreibenden Motors gekoppelten Drehzahlsensor kann die Wagengeschwindigkeit ermittelt werden. Die Kennlinie dieses Sensors ist ebenfalls linear. Um den Winkel, den das Pendel zur Senkrechten hat, zu ermitteln, wird ebenfalls ein Sensor mit linearer Kennlinie verwendet. Das Pendel ist dabei auf einen Bereich von $\pm 30°$ mechanisch durch Stoppelemente beschränkt. Eine ausführliche Auseinandersetzung mit den Kennlinien der verwendeten Sensoren sowie der Signalskalierung ist in [8], S.15ff. zu finden.

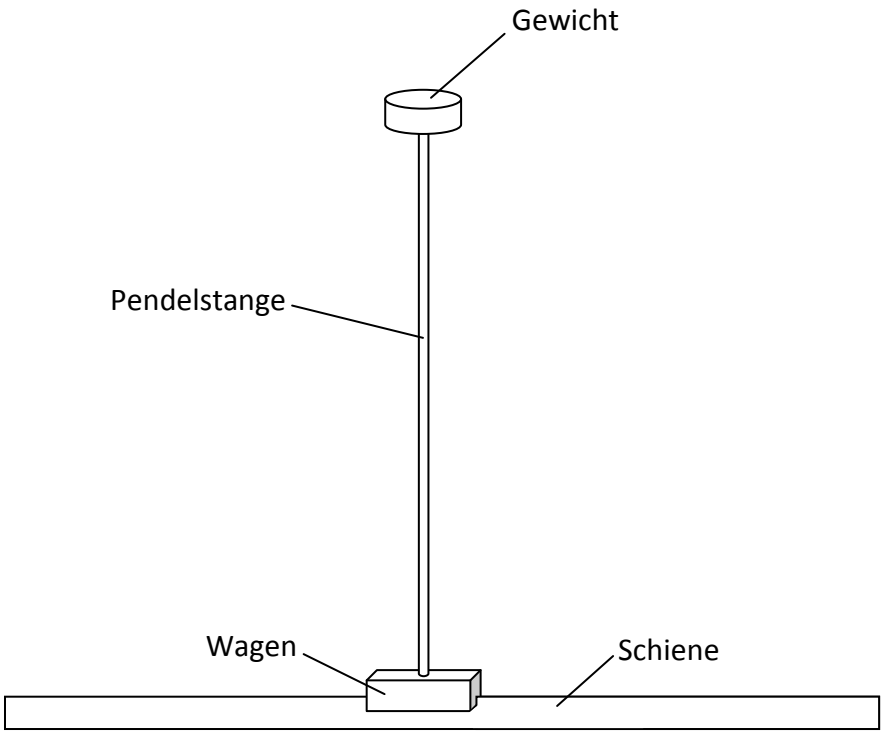


Abbildung 15: Versuchsaufbau des Pendels - Schema

3.5.1.2 MATLAB/Simulink, dSPACE-Box und ControlDesk

Die von den Sensoren des Pendels ermittelten Daten werden über ein Verbindungskabel an die dSPACE-Box weitergeleitet. Auf dieser befinden sich die kompilierten Daten, die zuvor in MATLAB/Simulink erstellt wurden. Die Verwendung des Real-Time Workshop von MATLAB

ermöglicht die schnelle und effiziente Erzeugung eines lauffähigen C-Codes aus den erstellten Simulink-Modellen. Dazu muss zunächst das gewünschte Modell geöffnet und die nötigen Parameter geladen werden. Anschließend startet man über den Befehl „Build Model“ die Generierung des C-Codes auf dem Rechner und das Programm wird auf der RCP-Umgebung abgelegt. Die Speicherung auf der dSPACE-Box erfolgt via Verbindungskabel, ebenso die gesamte Kommunikation des Anwender-PCs mit der Box. Wird das Modell geändert (neue Gewichte sollen verwendet werden), muss das Programm neu kompiliert werden. Beim Kompilieren wird außerdem eine „**.sdf*“-Datei generiert, welche die gleiche Bezeichnung wie das genutzte Modell trägt. Diese Datei kann dazu genutzt werden, auf anschauliche Weise die aktuellen Messwerte des Systems darzustellen. Dazu muss sie im dSPACE Programm „ControlDesk“ in ein Experiment geladen werden. Sie beinhaltet die Variablenstruktur des erstellten Simulink-Modells. Einmal eingebunden können die Variablen des Systems angezeigt und auch verändert werden. Auf diesem Weg kann beispielsweise die Vorgabe einer veränderlichen Sollposition für den Wagen realisiert werden. Außerdem kann somit sehr anschaulich der zeitliche Verlauf aller Zustände des Systems dargestellt werden.

3.5.2 Umsetzung

Um das zur Simulation erstellte Modell am realen Pendel einsetzen zu können, muss es dahingehend modifiziert werden, dass die benötigten Eingangssignale der beiden künstlichen neuronalen Netze, also die Zustände des Pendels, nicht mehr vom Modell, sondern durch die Sensoren des Systems bestimmt werden. Einen Überblick über das neue Modell gibt Abbildung 16.

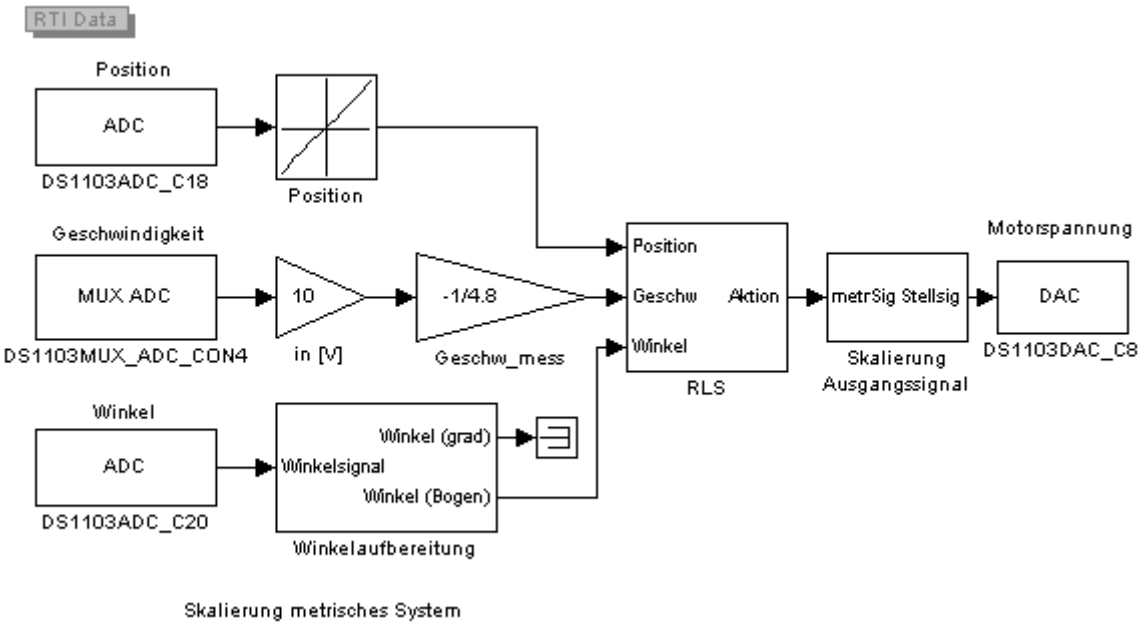


Abbildung 16: Simulink-Modell *„RLS_KNN_reales_Pendel.mdl“*

Der Aufbau der Kernkomponente des RLS bleibt weitestgehend unverändert. Die beiden KNN sind mit der Einheit zur Fehlersignalgenerierung verbunden, welche in diesem Fall jedoch nur zum Adaptieren der Gewichte genutzt wird, nicht aber zum Zurücksetzen des Pendelmodells. Im Fehlerfall muss das Pendel von Hand zurückgesetzt werden. Auch das KNN_WF besitzt in diesem Fall nur dann eine Funktion, wenn ein weiteres Lernen erfolgen soll. Abbildung 17 zeigt den neuen Aufbau des RLS.

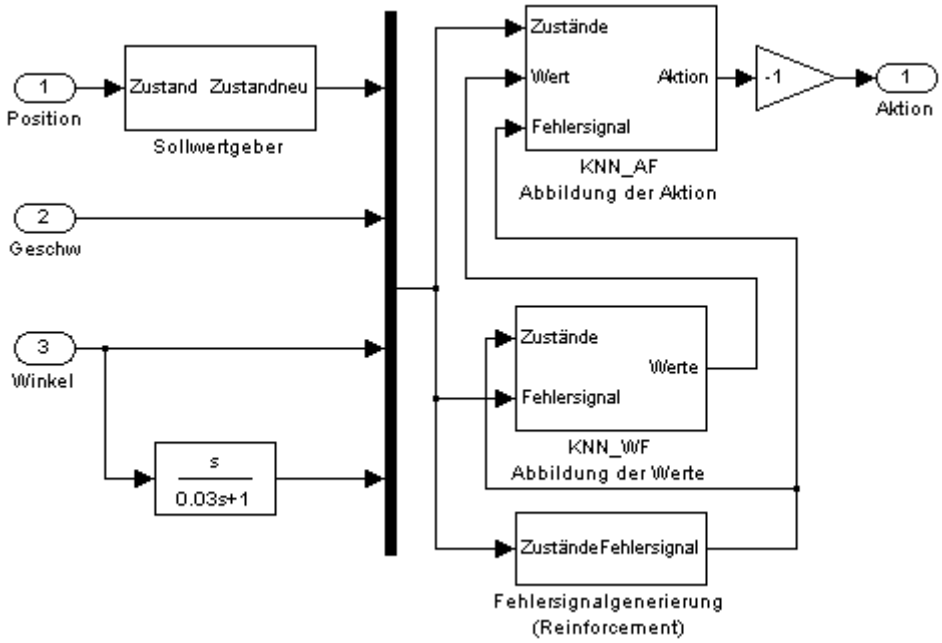


Abbildung 17: Die Hauptkomponente des kompletten RLS am realen Pendel

Da das Lernen am Pendel schwierig zu realisieren ist, wurde eine Variante des Modells erstellt, die im Kern nur aus dem KNN_AF besteht. Auch der Block zur Fehlersignalgenerierung wurde in dieser Version entfernt. Wie dieses Modell aufgebaut ist, illustriert Abbildung 18.

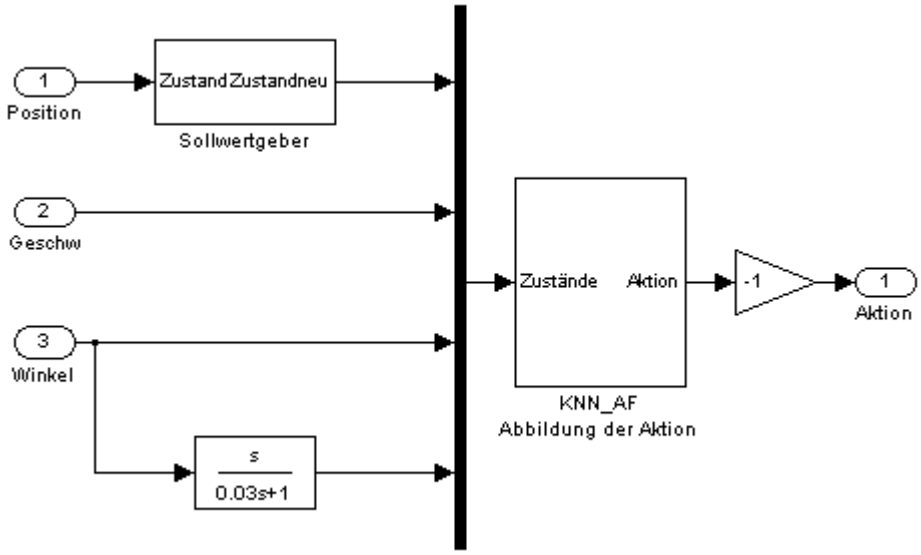


Abbildung 18: Die Hauptkomponente des RLS am realen Pendel ohne KNN_WF und Block zur Fehlersignalgenerierung

Zunächst wird der Einsatz des Modells zum Stabilisieren des Pendels untersucht. Dazu müssen im Vorfeld mit Hilfe des Modells *„RLS_KNN_Simulation.mdl“* die Gewichte zumindest für das KNN_AF angelernt werden (siehe Kapitel 3.3.3). Diese Gewichte werden mit Hilfe der Datei *„Gewichtsuebergabe.m“* als Startwert an die Memory-Blöcke des Modells *„RLS_KNN_reales_Pendel.mdl“* übergeben. Außerdem ist der Lernparameter des KNN_AF dieses Modells mit $\varepsilon = 0$ zu wählen, wodurch eine Veränderung der Gewichte im laufenden Prozess vermieden wird. Auch bei diesem Simulink-Modell müssen alle Startwerte des Systems initiiert werden (Datei *„Initialisierung.m“*). Das Modell ist anschließend wie in Kapitel 3.4.1.2 beschrieben zu Kompilieren und das Pendel kann in Betrieb genommen werden.

Kapitel 4
Darstellung und Interpretation der Ergebnisse

Die Ergebnisse aus der vorgestellten Untersuchung sind Gegenstand dieses Kapitels. Die zweigeteilte Betrachtung des Problems (Simulation und Anwendung am realen Pendel) soll aus Gründen der Verständlichkeit und aufgrund der strukturellen Unterschiede auch beim Evaluieren der Lösung beibehalten werden. Deshalb wird zunächst eine Betrachtung und Einschätzung der in der Simulation erzielten Ergebnisse vorgenommen. Dazu werden die durch die beiden künstlichen neuronalen Netze erlernten Funktionen untersucht und dargestellt sowie ihre gegenseitige Beeinflussung aufgezeigt. Abschließend wird die Güte der Simulation abgeschätzt.

Im Anschluss daran werden die Ergebnisse der Anwendung am realen Pendel untersucht. Dazu wird abgeschätzt, ob und wie die in der Simulation gewonnen Daten übertrag- und anwendbar sind. Weiterführend wird die Qualität der Regelung des realen Pendels mit Hilfe dieser Daten untersucht und evaluiert. Abschließend werden Mängel des Verfahrens aufgezeigt.

4.1 Ergebnisse der Simulation

4.1.1 Das KNN_WF

Der Lernverlauf in diesem Netz ist zu großen Teilen von den verwendeten Lernparametern α, β, γ, ε und ω abhängig. Dabei beeinflussen die Parameter β (Parameter des Momentum-Terms, vgl. Kapitel 2.3.3.3) und ω (Parameter zur Veränderung der Ableitung, vgl. Kapitel 2.3.3.4) im Wesentlichen die Geschwindigkeit der Konvergenz des Verfahrens. Sie sind Algorithmen zur Steigerung der Konvergenzrate und zur Minimierung des zeitlichen

Aufwandes (vgl. [3], S.181 und S.187ff.) und konnten mit der Werten $0{,}5 \leq \beta \leq 0{,}9$ und $0{,}1 \leq \omega \leq 0{,}2$ optimal eingesetzt werden. Auch die Parameter α und γ (beides Parameter der Reinforcement Learning Algorithmen, vgl. Kapitel 2.2.4) besitzen eine ähnliche Wirkung auf den Lernprozess, haben aber außerdem noch einen starken Einfluss auf die Genauigkeit der Schätzung der Wertefunktion. Der Parameter ε steht für die Schrittweite des Lernverfahrens und wurde mit Werten zwischen $0{,}01 \leq \varepsilon \leq 0{,}1$ initiiert.

Die Änderungen in der Aktionsfunktion zwischen zwei Fehlerfällen sind relativ klein, woraus häufig eine Folge von ähnlichen aufeinanderfolgenden Zustandsverläufen resultiert. Demzufolge sind auch die Fehlerursachen dieser Zustandsfolgen gleich. Zu Beginn des Lernprozesses wurde das KNN_AF beispielsweise so initiiert, dass die generierte Aktion das Pendel in den häufigsten Fällen nach rechts umkippen lässt. Dadurch wird zunächst garantiert, dass diese Fehlerfolge sehr oft wiederholt wird und das KNN_WF in dieser Zeit die korrekten Zustandswerte für diesen Bereich des Zustandsraumes erlernen kann. Es findet also eine Exploration in diesem Bereich des Zustandsraumes statt. In Abbildung 19 wird diese Erforschung für einen typischen Fall dargestellt.

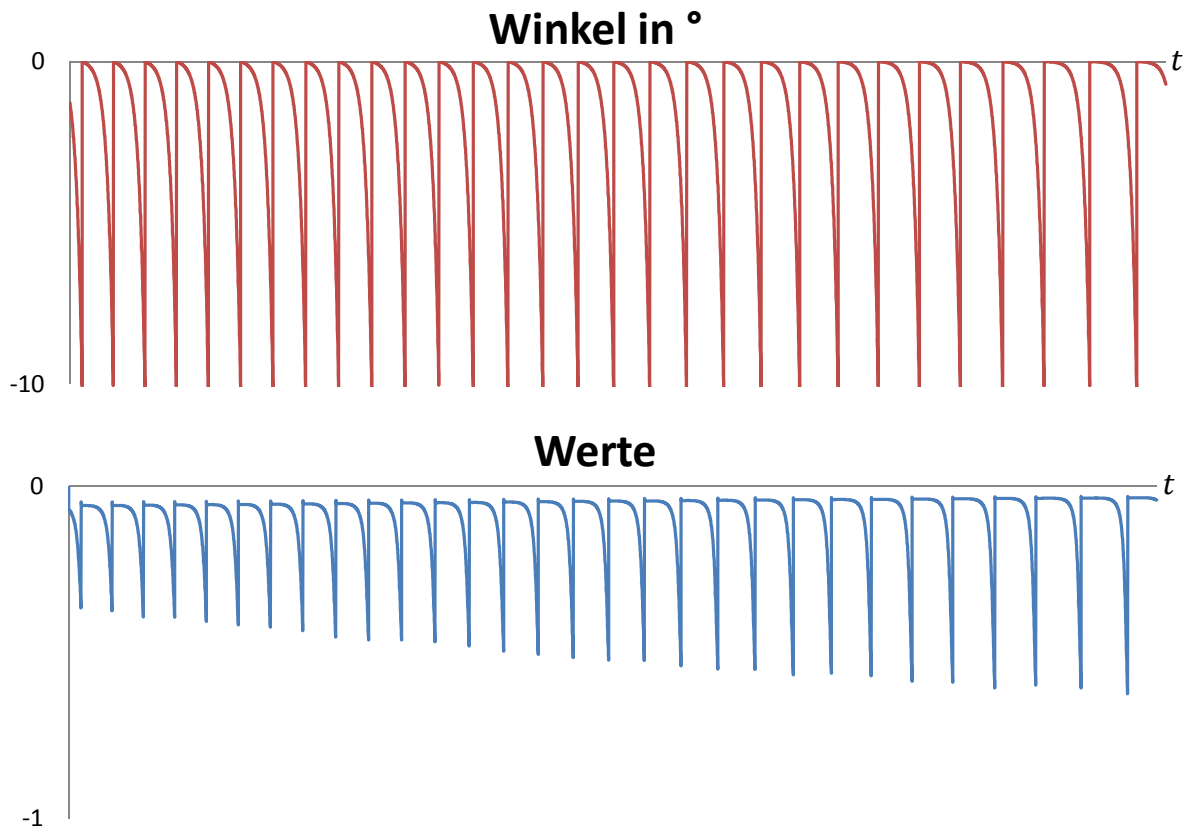


Abbildung 19: Typische Werteentwicklung bei Erforschung des Zustandsraumes bezüglich des Zustandes x_3

Im Laufe des Lernprozesses zeigte sich, dass das Anlernen der kompletten Wertefunktion nicht notwendigerweise stattfinden muss, um eine relativ genaue Approximation der Aktionsfunktion zu gewährleisten. Dies liegt vor allem daran, dass das KNN_WF bei günstig gewählten Werten im Besonderen für den Parameter α die Werte der aktuellen Zustandsserie sehr schnell erlernen kann. Dieser Parameter bestimmt, zu welchen Teilen der Unterschied zwischen aktuellem Zustandswert + Reinforcement und dem vorherigen Zustandswert in die Schätzung des neuen Wertes des vorherigen Zustandes eingeht. Er bestimmt also, wie sehr sich zwei direkt aufeinanderfolgende Zustandswerte ähneln sollen. Bei einer geringen Anzahl an möglichen Zuständen führt die Wahl eines großen Wertes für α zu starken Schwankungen in ihren Werten sobald die Wertefunktion ausgelernt ist. Da im Fall des Pendels eine Vielzahl an Zuständen und eine relativ genaue Abtastrate verwendet wurde, kann jedoch ein großes α nahe 1 verwendet werden, was den Prozess des Erlernens erheblich beschleunigt. Das Auftreten des Reinforcementwertes von -1 im Fehlerfall wird so zu starken Anteilen zum Update des Vorgängerzustandes verwendet und dieser wiederum zu großen Teilen zur Neuberechnung für seinen Vorgängerzustand usw. genutzt.

Durch die Wahl eines großen α nahe 1 und durch das häufige Auftreten gleicher Fehlerfälle wird ein schnelles Erlernen der Werte im Bereich des aktuellen Zustandsraumes garantiert. Die Aktionsfunktion kann somit schnell die korrekten Werte der Wertefunktion nutzen um für diesen Bereich des Zustandsraumes die korrekten Aktionen zu erlernen. Treten alternierende Zustandsverläufe auf (z. B. abwechselndes Umkippen des Pendels nach links und rechts), erlernt das Netz aufgrund seiner Struktur (Hiddenschicht) auch diesen nichtlinearen Zusammenhang. Dieser Lernschritt ist jedoch stark abhängig von den Parametern α und ε. Es zeigte sich, dass bei der Wahl kleiner Werte für diese beiden Konstanten eine sehr langsame Konvergenz hin zu den hinreichend genauen Werten zu beobachten ist.

4.1.2 Das KNN_AF

In diesem Netz finden die Parameter β, ε und ω Anwendung und sollten wie in Kapitel 4.1.1 beschrieben initiiert werden. Der Wert für ε sollte jedoch erfahrungsgemäß sehr groß

gewählt werden. Im Fall des KNN_AF ergab sich $\varepsilon = 10$ als praktikabler Wert. Nachdem die Gewichte des Netzes mit Werten zwischen ± 1 initiiert wurden, lernt es diejenigen Aktionen den jeweiligen Zuständen zuzuordnen, welche das Auftreten eines Fehlers vermeiden. Da das Pendel zunächst sehr viel häufiger umkippt, als dass der Wagen in die seitlichen Begrenzungen fährt, wird das KNN_AF zuerst erlernen, diesen Fehler zu vermeiden. Resultierend daraus kann eine Oszillation des Winkels um $0°$ beobachtet werden. Die Abweichung vom Ruhepunkt hängt von der maximalen Stellamplitude und der Dynamik des Pendels, also den verwendeten Massen und der Länge des Pendelstabes und den Lernparametern ab. Abbildung 20 zeigt einen für diesen Fall typischen Verlauf des Zustandes x_3 und die getätigten Aktionen.

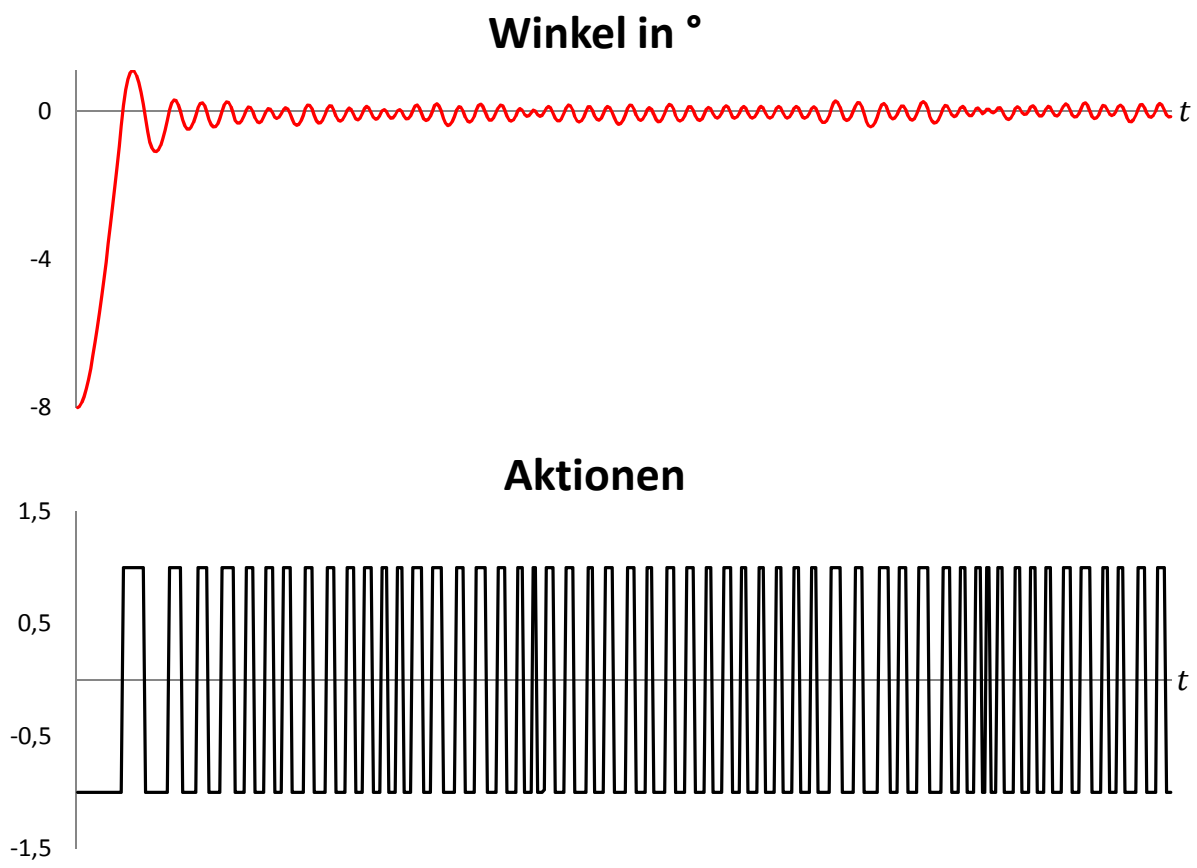


Abbildung 20: Typischer Verlauf der Stabilisierung des Pendels

Der nun auftretende Fehler des Fahrens des Wagens in die seitlichen Begrenzungen hat eine erneute Exploration des Zustandsraumes zur Folge (siehe Abb. 21). Es können große Werte

für die Position des Wagens beobachtet werden, welche zuvor durch das schnelle Umfallen des Pendels und das anschließende sofortige Rücksetzen vermieden wurden.

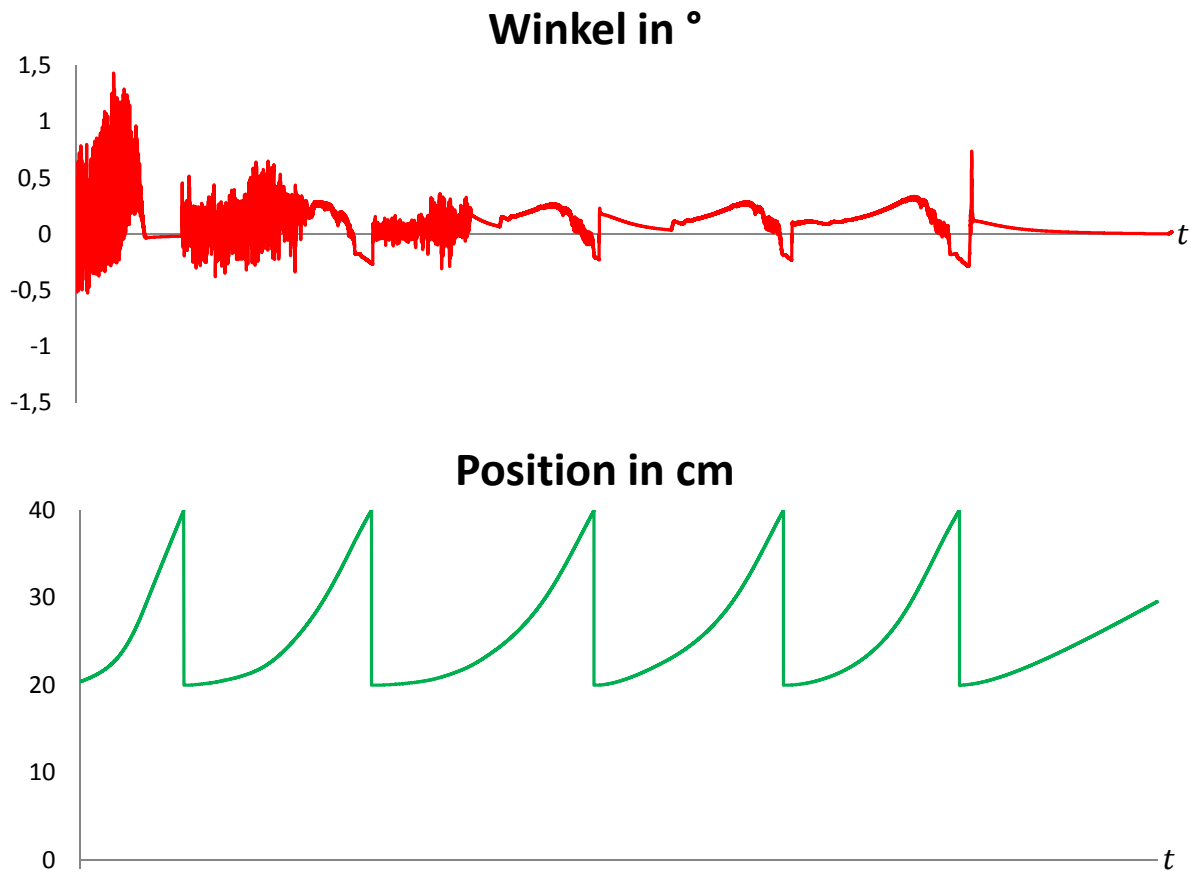


Abbildung 21: Typische Verläufe der Zustandes x_1 und x_3 nach erfolgreichem Erlernen des Ausbalancierens

Durch diese neuerliche Exploration und die damit verbundenen Fehlerfälle des Fahrens in die seitlichen Banden lernt die Wertefunktion die hinreichend korrekten Werte bezüglich der Zustände x_1 und x_2 abzubilden. Auf diese Weise kann auch die Aktionsfunktion die korrekten Aktionen für diesen Zustandsbereich erlernen. Diese Aktionen müssen so gewählt werden, dass sie den Wagen nicht in die Begrenzung fahren lassen, also eine bezüglich der aktuellen Position umgekehrte Bewegung hin zur Sollposition erwirken und gleichzeitig das Pendel nicht umkippen lassen. Wie dies erreicht werden kann, ohne das Pendel dabei umfallen zu lassen, zeigt Abbildung 22.

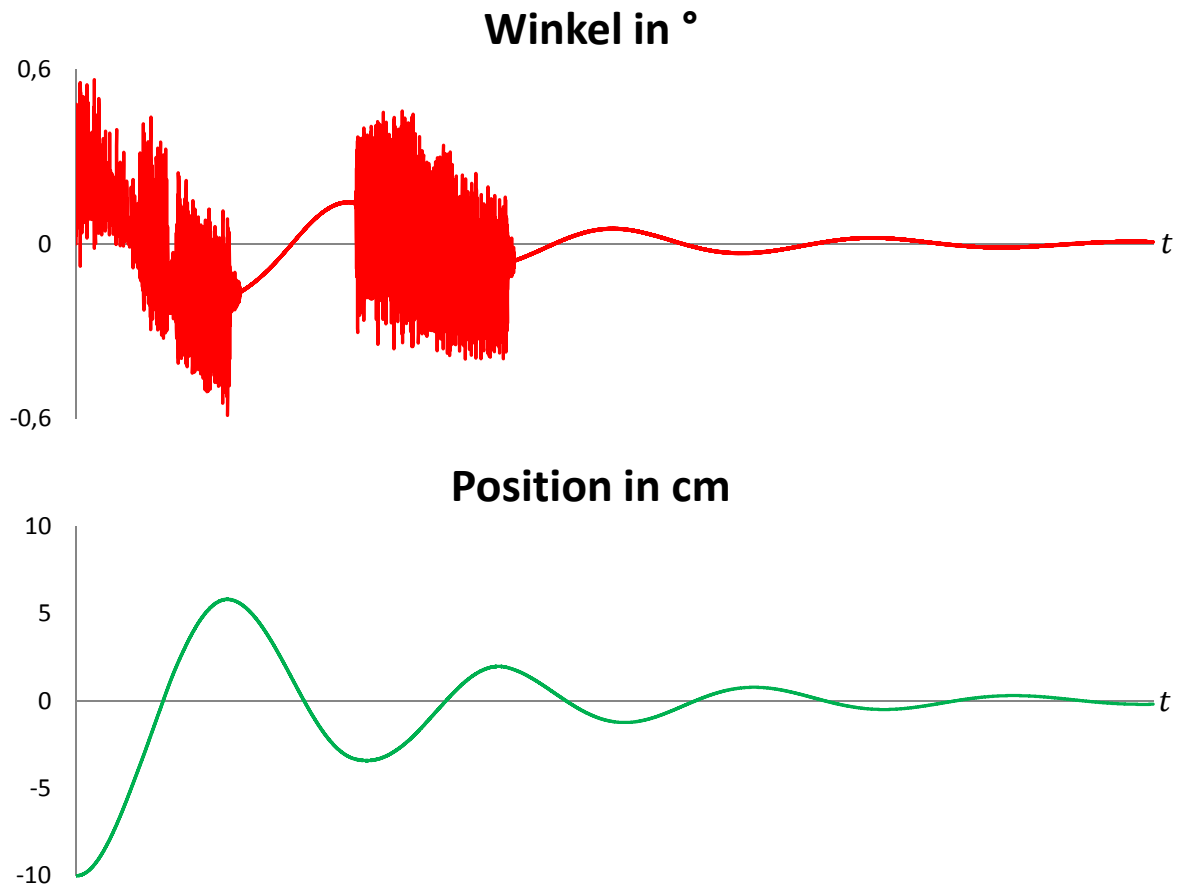


Abbildung 22: Regelung der Position des Wagens mit ausgelernter Aktionsfunktion

Die Position des Wagens wird, wie auch die Auslenkung des Winkels, aufgrund der Trägheit des Systems sinusförmig mit kleiner werdender Amplitude in die Ruheposition (Nulllage) zurückgebracht. Dieser Sinus findet sich auch im Grafen des Verlaufs des Zustandes x_3 wieder. Auf diese Weise erklärt sich auch, wie der Algorithmus die gleichzeitige Stabilisierung beider Zustände realisiert. Der Winkel wird mit einem Offset belegt, welcher dazu dient, den Wagen in die jeweilige Richtung fahren zu lassen.

4.1.3 Einhaltung der Sollposition in der Simulation

Nachdem die Aktionsfunktion die Regelung beider Prozesse erlernt hat, kann eine Untersuchung hinsichtlich der Verfolgung einer Sollposition vorgenommen werden. Dazu

werden die in Kapitel 3.4 vorgestellten Algorithmen verwendet. Die Auswertung dazu findet sich in Abbildung 23.

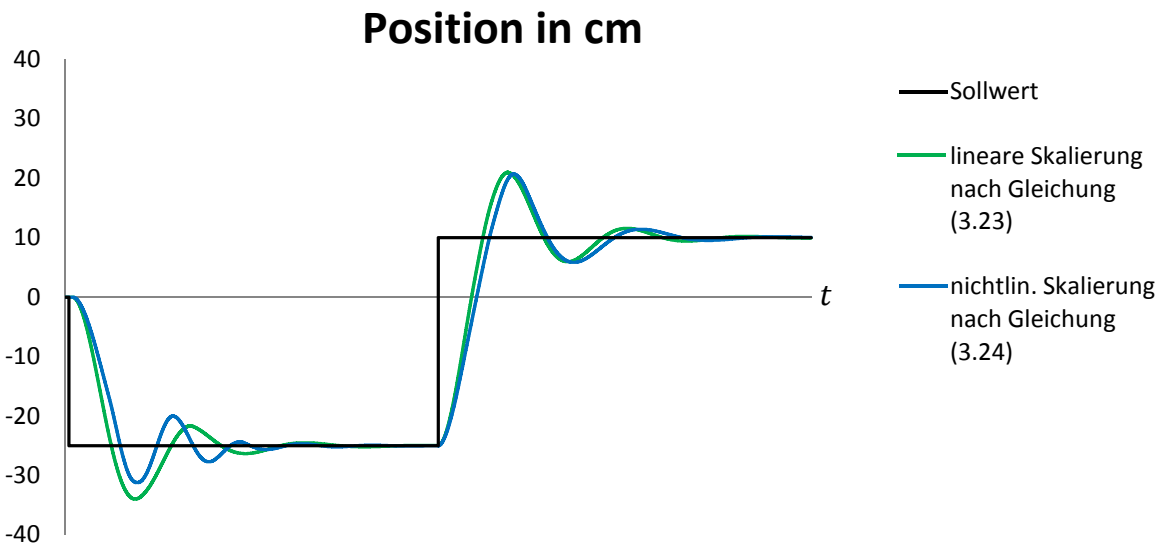


Abbildung 23: Sollwertverfolgung mit Hilfe verschiedener Skalierungsmethoden

Die Varianten erweisen sich insgesamt als praktikabel. Sollwertvorgaben, die nahe dem Grenzbereich der Position liegen, werden durch die nichtlineare Skalierung jedoch deutlich sicherer abgefangen (geringeres Überschwingen). Deshalb ist für große Sprünge diese Methode zu empfehlen.

4.1.4 Güte der Simulation

Die erreichten Ergebnisse erfüllen die an das Verfahren gestellten Erwartungen. Es konnten verschiedene Gewichtskombinationen für das Netz ermittelt werden, welche die gestellte Aufgabe, die Stabilisierung des Pendels und die Sollpositionierung des Wagens, lösten. Die Zeit zur Ermittlung dieser Kombinationen variierte stark und ist abhängig von den eingesetzten Lernparametern.

Die gefundenen Lösungen beinhalteten jedoch alle ein Offset bezüglich der stationären Position des Wagens. Begründet liegt dies im Verfahren selbst. Die Aktualisierung der

Aktionen finden aufgrund der Wertänderungen TD zweier aufeinanderfolgender Aktionen statt (siehe Kapitel 2.2.6). Findet das Netz während eines Berechnungsschrittes eine Gewichtskombination, welche die Wagenposition konstant hält (Positionsänderung nur noch durch Pendelbewegung, vernachlässigbar klein), so findet durch die aufeinanderfolgenden Zustände keine Wertänderung mehr statt und damit auch keine Anpassung der Aktionen. Der Wert des Zustandes kann in diesem Moment noch immer relativ klein sein (durch stationäre Lagen von x_1 nahe ± 40), doch die Wertänderung TD ist Null und somit auch die Änderungen des Ausgangs des KNN_AF bzw. die an der zurückliegenden Aktion vorgenommene Änderung. Dieses „Einschwingen“ des Zustandes x_1 um einen Wert, der nicht der Nulllage entspricht, kann beim Winkel nicht beobachtet werden, da das Netz in diesem Fall aufgrund der aufgewendeten Energie in die seitliche Begrenzung fährt und der somit auftretende Fehlerfall es zum Weiterlernen zwingt. Dafür ist die Dynamik des Pendels verantwortlich. Diese Abweichung muss beim Anwenden der gefundenen Gewichtskombination beachtet werden. Eine zusätzliche Vorskalierung des Zustandes x_1 kann diese Problematik beseitigen.

Ein weiteres Problem ist die Tatsache, dass es sehr viele verschiedene Lösungen für die Zielstellung gibt. Beachtet man Abbildung 21 genau, lässt sich feststellen, dass das Netz gelernt hat, die Auslenkung des Wagens zum Teil mit einem stark verrauschten Sinus zu beseitigen (Gewichte gespeichert in Datei „*Gewichte2.mat*“). Die Anwendung dieses Parametersatzes löst zwar das gestellte Problem, bedeutet in der Anwendung am Versuchsaufbau jedoch eine starke alternierende Beanspruchung des Antriebes des Systems. Ähnliche Beobachtungen konnten bei anderen Gewichtskombinationen, die eine Lösung für das Problem lieferten, gemacht werden. Abbildung 24 zeigt den zeitlichen Verlauf der Position des Wagens und des Winkels des Pendels nach einer initialen Auslenkung des Wagens (vgl. Abb. 22), die mit einer alternativen Gewichtskombination realisiert werden konnte. Man erkennt, dass diese Lösung bezüglich des Zustandes x_1 kein Überschwingen provoziert und auch der Zustand x_1 schwingt nur sehr wenig, was eine geringere alternierende Belastung des Stellgliedes zur Folge hat. Diese Lösung ist dementsprechend derjenigen vorzuziehen, die zur Generierung der zeitlichen Verläufe in Abbildung 22 verwendet wurde. Die verwendete Gewichtskombination wurde in der Datei „*Gewichte1.mat*“ hinterlegt.

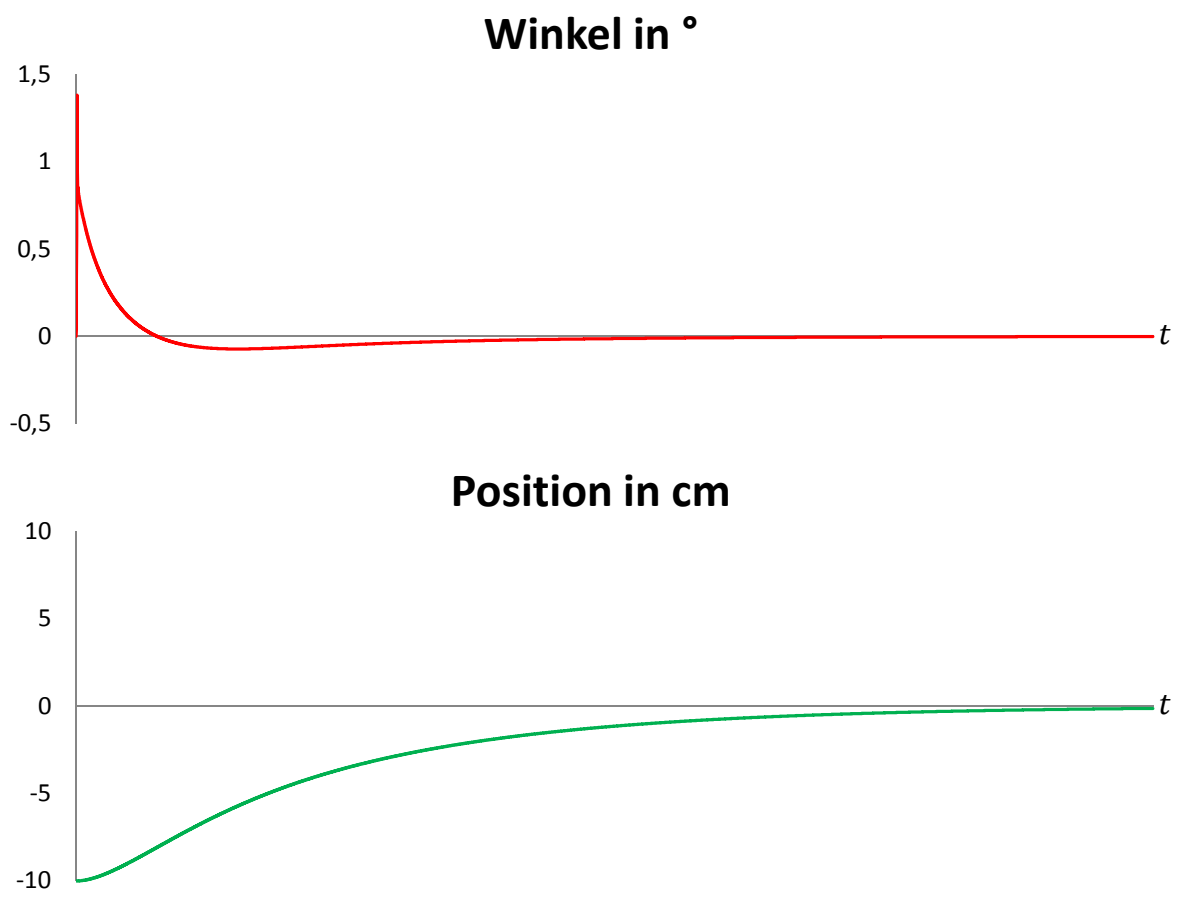


Abbildung 24: Regelung der Position des Wagens mit ausgelernter AF und alternativer Gewichtskombination

Ist die Wertefunktion einmal angelernt und die beiden möglichen Fehler werden vermieden, ist zu beobachten, dass das KNN_WF die realen Werte der Zustände „vergisst“. Tritt ein Fehler auf, spiegelt sich dies nicht in den Werten wieder, die realen Werte müssen erneut erlernt werden. Die Ursache dieser Problematik liegt in der Generierung der Werte. Sie hängen davon ab, ob ein Fehler auftritt, oder nicht (vgl. Kapitel 2.2 und Kapitel 3.3.1). Findet man z. B. eine Gewichtskombination für das KNN_AF, die eine sinusförmige Stabilisierung des Pendels realisiert, welche eine grenzwinkelnahe Amplitude hat, treten trotz großer Winkel keine Fehler auf. Zunächst hatte das KNN_WF jedoch gelernt, dass große Winkel einen kleinen Zustandswert besitzen, da sie höchstwahrscheinlich ein Umkippen des Pendels zur Folge haben. Durch das Fehlen des durch den Fehlerfall verursachten negativen Rewards $r = -1$, nähern sich die Zustandswerte auch für große Winkel dem Wert Null, genau dem Wert, der als Reward dient, wenn kein Fehler auftritt. Im Unendlichen konvergiert die

Wertefunktion für alle Zustände zu Null. Wie schnell dies geschieht hängt von den verwendeten Lernparametern ab.

4.2 Ergebnisse reales Pendel

Nachdem die Gewichte für die Aktionsfunktion hinreichend gut mit Hilfe der Simulation angelernt wurden, können sie in das Modell „*RLS_KNN_reales_Pendel.mdl*“ geladen werden und der Aufbau kann auf Stabilität und Positionseinhaltung getestet werden. Im Rahmen dieser Arbeit wurden mehrere Gewichtskombinationen, die in der Simulation für beide Ziele (Stabilisierung Pendel, Sollpositionierung) positive Ergebnisse erzielten, am Versuchsaufbau getestet. Ein Teil der Simulationslösungen konnte nicht erfolgreich am realen Versuchsaufbau eingesetzt werden. Die Stabilisierung des Pendels gelang zunächst in den meisten Fällen, doch die Sollpositionierung erwies sich häufig als instabil. Ein Aufschwingen war bezüglich der Sollposition nicht zu beobachten, der Fehler trat durch Fahren in die seitlichen Begrenzungen bzw. durch das nicht Einhalten der Sollposition auf.

Diejenige Kombination von Gewichten, die alle gestellten Regelungsziele auch am realen Prozess erfüllt, wurde in der Datei „*Gewichte1.mat*“ gespeichert. Nachdem die in ihr hinterlegten Gewichte in das Modell geladen wurden und dieses auf die RCP-Umgebung übertragen wurde, können die Zustandsverläufe am Pendel beobachtet und mit Hilfe des Programms ControlDesk aufgezeichnet werden. Es zeigte sich, dass das Pendel durch die Aktionsfunktion bezüglich des Zustandes x_3 (Winkel) sehr stabil gehalten werden kann, größere Winkel treten nicht auf. Abbildung 25 zeigt einen typischen zeitlichen Verlauf für diesen Zustand und korrespondierend dazu den Verlauf der Position des Wagens. Die Einhaltung der Position erfolgt mit einer Abweichung von $\approx \pm 10cm$ und das Pendel schwingt zwischen $\approx \pm 3°$. Die Abweichung der Position ist auf die auftretende Reibung (Slip-Stick-Effekt) zurückzuführen. Das Ziel der Beseitigung des Effektes konnte nicht erreicht werden. Begründet liegt dies in der Tatsache, dass diese Nichtlinearität nicht modelliert wurde und somit während der Simulation (Anlernen der Gewichte) nicht beachtet werden konnte.

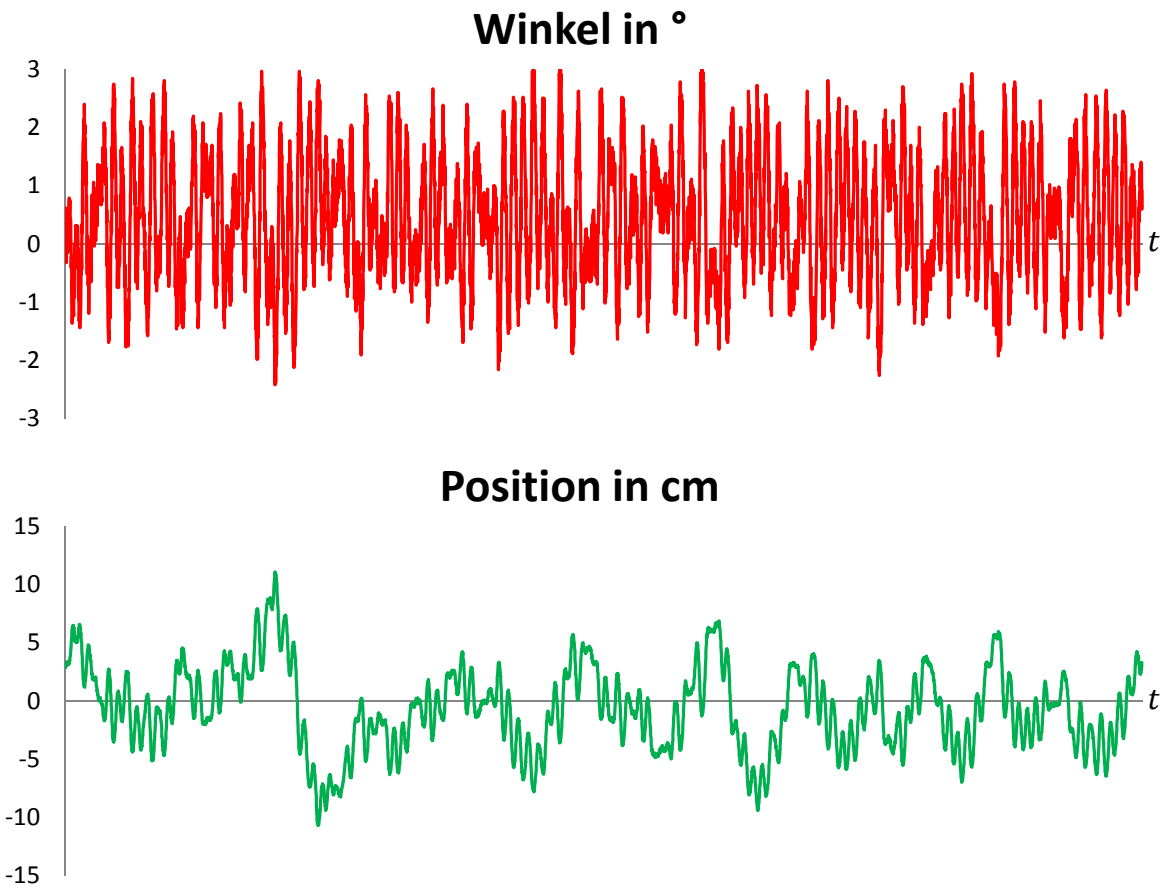
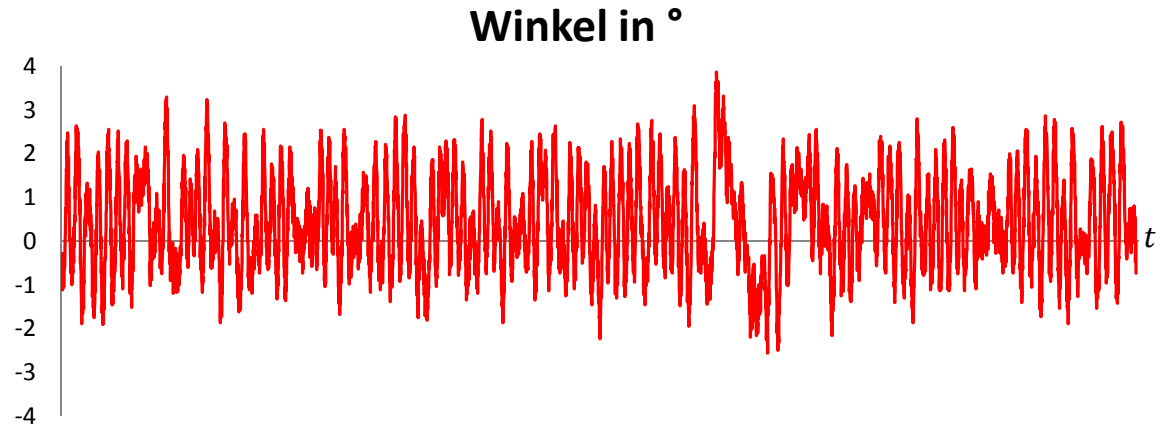


Abbildung 25: Zeitlicher Verlauf der Zustände x_1 und x_3 am realen Prozess

Die Überführung des Wagens in die Sollposition wird in Abbildung 26 dargestellt. Dabei erfolgt ein Sprung von $30cm$ von der Sollposition $x_1^{Soll} = -20$ auf $x_1^{Soll} = 10$. Es ist zu erkennen, dass zum Zeitpunkt des Sprungs der Winkel der Pendelauslenkung zunächst vergrößert wird um die Umsteuerung zur neuen Sollposition zu ermöglichen. Folgend wird er solange verringert, bis die Sollposition erreicht wurde, um danach wieder in die stabile Schwingungslage überführt zu werden.



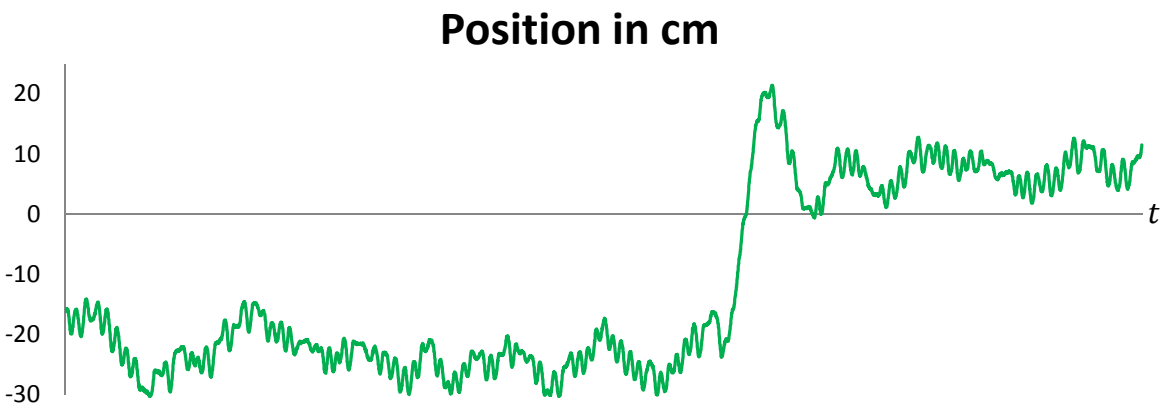


Abbildung 26: Zeitliche Verläufe der Zustände x_1 und x_3 bei einem Sollwertsprung von $x_1^{Soll} = -20$ auf $x_1^{Soll} = 10$

4.3 Vergleich Simulation und realer Prozess

Um die Abweichungen zwischen den Ergebnissen der Simulation und der Anwendung am realen Modell unter der Verwendung identischer Gewichtskombination für das KNN_AF erklären zu können, muss man sich vor Augen führen, wo die Unterschiede zwischen dem Modell und dem Versuchsaufbau liegen. Die größte Differenz zwischen beiden ist die durch die Reibung auftretende Nichtlinearität (Reibung, Slip-Stick-Effekt), welche im Modell nicht berücksichtigt wird. Durch die Anpassung des Modells z. B. durch das Einführen einer Verzögerung der Aktion (entspricht der durch Reibung und Trägheit verzögerten Antwort des Wagens auf eine Spannungsänderung der Stellgröße) können ähnliche zeitliche Verläufe bei Sollwertänderungen erzielt werden, wie sie am Versuchsaufbau beobachtet werden können.

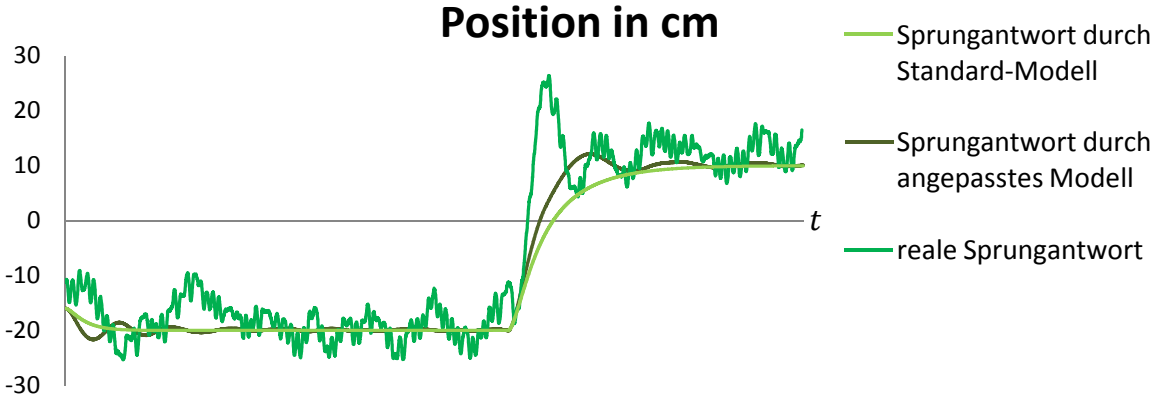


Abbildung 27: Vergleich zwischen realem Zustandsverlauf und dem Verlauf beim angepassten und beim Standard-Modell

Kapitel 5
Vergleich mit vorhandenen Regel-algorithmen

In diesem Kapitel wird der bereits realisierte Ansatz (Zustandsregler) zur Regelung des inversen Pendels mit dem in dieser Arbeit vorgestellten Algorithmus verglichen. Dazu wird die Realisierung einer Zustandsregelung zunächst kurz vorgestellt und anschließend ein Vergleich beider Vorgehensweisen bezüglich der Stabilität der Winkelauslenkung und der Erreichung und Einhaltung von Sollwerten (Wagenposition) vorgenommen.

5.1 Vorstellung und Evaluierung vorhandener Algorithmen

Ein Zustandsregler nutzt die inneren Zustände eines Systems um es zu regeln, indem er alle Zustände im Regelkreis zurückführt. Diese Regelung kann auf verschiedene Arten erfolgen, z. B. durch Polvorgabe oder mit Hilfe des Ricatti-Ansatzes. Die Voraussetzungen zur Anwendung dieser Verfahren ergeben sich aus ihrer Arbeitsweise. Zunächst müssen alle Zustände beobachtbar sein. Können sie nicht gemessen werden, müssen Zustandsbeobachter eingesetzt werden. Zum anderen müssen sie steuerbar sein. Die Zustände werden jeweils mit einem Proportional-Element multipliziert und anschließend zurückgeführt (vgl. [9], S.644ff.).

Der vorliegende Zustandsregler wurde mit Hilfe des Ricatti-Ansatzes erstellt. Bei der Verwendung dieses Ansatzes wird ein während des Entwurfes zu minimierendes, quadratisches Gütemaß verwendet.

$$J = \frac{1}{2} \cdot \int (\boldsymbol{x}^{\boldsymbol{T}} \cdot \boldsymbol{Q} \cdot \boldsymbol{x} + \boldsymbol{u}^{\boldsymbol{T}} \cdot \boldsymbol{R} \cdot \boldsymbol{u})\, dt \tag{5.1}$$

Innerhalb dieses Gütemaßes erfolgt eine „Bewertung“ des Zustandsvektors mit der Matrix $\boldsymbol{Q}$. Die Steuergröße wird mit der Matrix $\boldsymbol{R}$ bewertet. Diese Bewertung stellt eine Art der Gewichtung dar. Die Koeffizienten dieser Matrizen werden je nach Zielstellung gewählt und

wichten im Fall der Matrix $\boldsymbol{Q}$ je einen Zustand. Hat die Regelung beispielsweise eine geringe Abweichung des Winkels von seiner Ruheposition zum Ziel, sollte der diesen Zustand wichtende Koeffizient relativ groß im Verhältnis zu den anderen Koeffizienten gewählt werden. Eine die Stellgröße betreffende Zielstellung wird mit Hilfe der Matrix $\boldsymbol{R}$ implementiert. Minimiert man dieses Gütemaß, ergibt sich die Reglermatrix $\boldsymbol{K}$, welche nach der Formel

$$\boldsymbol{u} = -\boldsymbol{K} \cdot \boldsymbol{x} \tag{5.2}$$

die jeweils aktuelle Stellgröße bestimmt (vgl. [8], S.37ff.).

Die erstellte Lösung bietet eine stabile Lösung die Pendelauslenkung betreffend. Auch die Einhaltung der Sollposition kann durch diesen Ansatz erreicht werden. Eine weiterführende Betrachtung ist in [8], S.37ff. zu finden.

5.2 Vergleich

Im Folgenden sollen die Anwendung des Zustandsreglers und die Verwendung des RLS zur Regelung des realen Pendels miteinander verglichen werden. Dazu wird zunächst eine Untersuchung hinsichtlich der Auslenkung des Pendels (Zustand x_3) bei stationärem Sollwert untersucht (Abb. 28 und Abb. 29). Anschließend werden die zeitlichen Verläufe der Zustände x_1 und x_3 für beide Verfahren nach Sollwertsprüngen aufgezeigt und interpretiert.

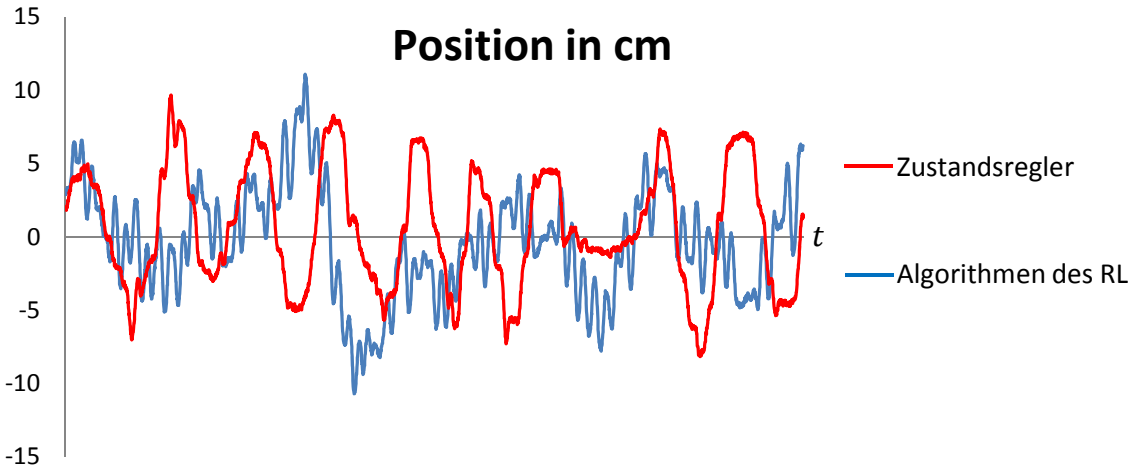


Abbildung 28: Zeitlicher Verlauf des Zustandes x_1 bei der Verwendung der verschiedenen Verfahren (fester Sollwert für x_1)

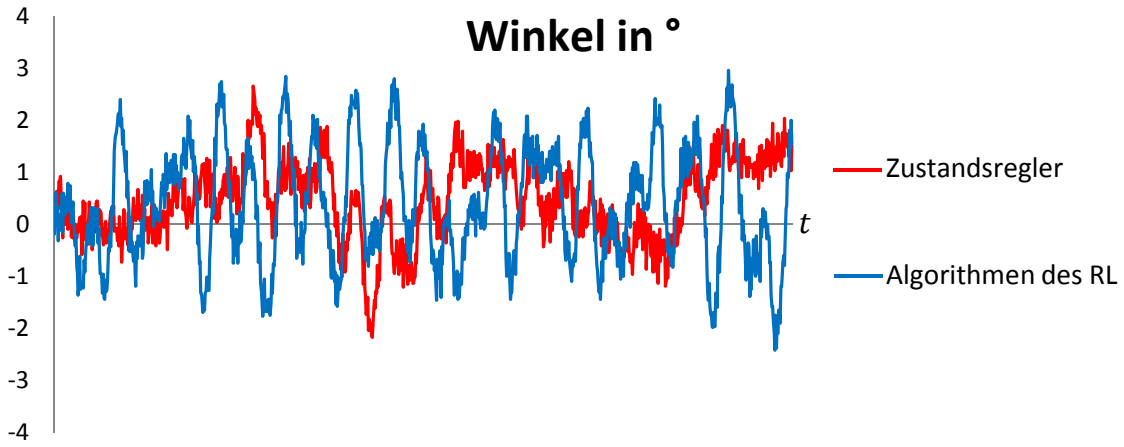


Abbildung 29: Zeitlicher Verlauf des Zustandes x_3 bei der Verwendung der verschiedenen Verfahren (fester Sollwert für x_1)

Die Einhaltung der Sollposition des Wagens erfolgt mit Hilfe des Zustandsreglers deutlich ruhiger als durch Algorithmen des Reinforcement Learning. Auch die maximale Regelabweichung ist bei letzteren z. T. größer als bei der Realisierung durch den Zustandsregler. Insgesamt sieht der Verlauf des Zustandes x_1 beim Zustandsregler sehr viel harmonischer aus, was auch darauf schließen lässt, dass das Stellglied weniger häufiger mit alternierender maximaler bzw. minimaler Stellamplitude beansprucht wird.

Bezüglich der Auslenkung des Pendels ist festzustellen, dass die Anwendung beider Verfahren eine maximale Auslenkung von $\approx \pm 2{,}5°$ hervorruft. Die Pendelbewegung, die bei der Regelung durch die RL Algorithmen hervorgerufen werden, scheinen dabei realtiv harmonisch zu sein. Der Zustandsregler hingegen versucht bereits kleine Abweichungden im Winkel zu beseitigen, woraus viele kleine Äderungen im Winkel resultieren (siehe Abb. 29).

Beide Vorgehensweisen eignen sich, um das Pendel stabil in seiner Lage zu halten. Legt man einen großen Wert auf die schonende Belastung des Antriebes des Wagens, sollte die Lösung mit Hilfe des Zustandsreglers bevorzugt werden, da dieser auch Stellwerte ausgibt, die nicht dem Maximal- bzw. Minimalwert der Stellgröße entsprechen.

Die Untersuchung der beiden Verfahren hinsichtlich der Anworten auf Sollwertsprünge der Position des Wagens wird in Abbildung 30 vorgenommen.

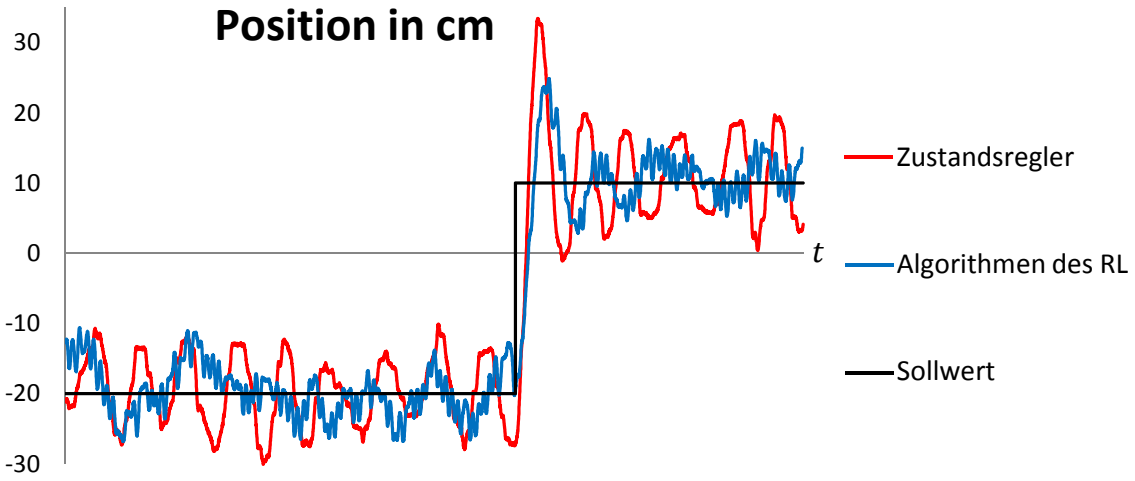


Abbildung 30: Zeitlicher Verlauf des Zustandes x_1 nach Sollwertsprung

Abbildung 30 zeigt, dass der vorgegebene Sollwert durch die Zustandsregelung schneller erreicht wird, als durch die Regelung mit Hilfe des RLS. Daraus resultieren jedoch auch ein deutlich größeres Überschwingen und eine größere Einschwingzeit. Insgesamt ist in diesem Fall der Einsatz der Algorithmen des RL vorzuziehen, um die Überführung des Wagens zu einer neuen Sollposition zu realisieren. Die Vorgabe zu großer Sollwertsprünge ($\Delta x_1^{soll} \geq 35cm$) resultiert beim Einsatz beider Verfahren sehr häufig im Umkippen des Pendels, zumeist durch das Fahren in die seitliche Begrenzung hervorgerufen. Die Sprünge sollten demzufolge nicht zu groß sein.

Kapitel 6
Fazit und Ausblick

Die in der Simulation erzielten Ergebnisse zeigen, dass es möglich ist ein inverses Pendel mit Hilfe der vorgestellten Algorithmen zu stabilisieren und die Position des Wagens, auf dem es steht, zu einer vorgegebenen Sollposition zu überführen. Dies geschieht quasi ohne a priori-Informationen über das zu regelnde Modell (Signalskalierung für die Netze ausgenommen). Es zeigte sich, dass der zeitliche Aufwand zur Adaption der Funktionen durch die Netze einen Wert überschreitet, der das Lernen direkt am Versuch unmöglich macht. Ein simulatives Erlernen der Aktions- und Wertefunktion ist demzufolge unumgänglich. Dabei muss bei der Verwendung des Modells zum Anlernen der Aktions- und Wertefunktion darauf geachtet werden, dass dieses im Vorfeld sehr genau geschätzt wird, da ein nachträgliches Adaptieren der Gewichte der verwendeten Netze an dem realen System aufgrund des Verfahrens selbst nur sehr bedingt bzw. gar nicht realisierbar ist.

Der Vergleich mit alternativen Lösungen zur Regelung des Pendels und zur Sollpositionierung des Wagens zeigt, dass die Algorithmen des RL konkurrenzfähig sind, um die gestellten Ziele effektiv zu erfüllen.

Das gestellte Ziel der Verminderung bzw. der Eliminierung der Nichtlinearität der Strecke, d. h. des Slip-Stick-Effektes, kann ohne Anpassung der Zielstellung für den Reinforcement Learning Algorithmus nicht erreicht werden. Um diese Aufgabe zu bewerkstelligen, muss zunächst eine Lösung für die Stabilisierung des Pendels und die Führung des Wagens, wie in der Arbeit beschrieben, gefunden werden. Anschließend muss eine neue Definition für den Fehlerfall der RLS-Algorithmen festgelegt werden. Diese Definition muss dem RLS-System genau dann einen Fehler melden (und damit einen negativen Reward zurückkoppeln), sobald der Slip-Stick-Effekt auftritt bzw. die aus diesem resultierenden, typischen Zustände beobachtbar sind, damit das RLS lernen kann, das Auftreten dieser Zustände in Zukunft zu vermeiden. Versucht man diese Konfiguration des RLS direkt am Versuchsaufbau anzulernen, treten für die Gewichtskombination nichtstabile Zwischenlösungen auf. Dadurch wird ein manuelles Zurücksetzen des Pendels erforderlich. Der damit verbundene zeitliche und manuelle Aufwand ist jedoch nicht zu bewerkstelligen, weshalb die Verwendung eines

Modells, das die betreffenden Nichtlinearitäten abbildet, zum Anlernen der hinreichend korrekten Aktions- und Wertefunktion notwendig wird.

Um diese Effekte zu relativieren und eine bessere Regelung zu erreichen, muss der Versuchsaufbau erneut modelliert werden. Um die Vorteile der Algorithmen des Reinforcement Learning nutzen zu können, muss dieses Modell sehr genau erstellt werden und die Nichtlinearität der Strecke muss in diesem enthalten sein.

Abkürzungsverzeichnis

AF – Aktionsfunktion

KNN – künstliches neuronales Netz

LMS – Least Mean Square

RCP – Rapid Control Prototyping

RL – Reinforcement Learning

RLS – Reinforcement Learning System

SuSy – Subsystem

TD – temporale Differenz

VF – Vorfilter

WF – Wertefunktion

ZR – Zustandsregler

Quellenverzeichnis

[1]: Alpaydin, E.: *Maschinelles Lernen*. Übers. Simone Linke. München: Oldenbourg Verlag, 2008.

[2]: Sutton, R. S.; Barto, A. G.: *Reinforcement Learning, An Introduction*. Cambridge, Mass. [u.a.]: MIT Press, 1998.

[3]: Patterson, D. W.: *Künstliche neuronale Netze : das Lehrbuch*, 2.Auflage. Haar bei München [u.a.]: Prentice Hall Verlag, 1997.

[4]: Adamy, J., Prof. Dr.-Ing.: *Fuzzy-Logik, Neuronale Netze und Evolutionäre Algorithmen*. Aachen: Shaker Verlag, 2007.

[5]: Rey, G. D.; Wender, K. F.: *Neuronale Netze : Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. Bern: Verlag Hans Huber, 2008.

[6]: Anderson, C.W.: *Learning to control an inverted pendulum with neural networks*. IEEE Control Systems Magazine, 9, No. 3, April, 1989, S. 31–36.

[7]: Schreiber, C.: *Reglerentwurf zur Stabilisierung eines inversen Pendels*. Ilmenau: Technische Universität Ilmenau, 2005.

[8]: Kulpe, A.: *Entwurf und Vergleich verschiedener Regler zur Stabilisierung eines inversen Pendels*. Ilmenau: Technische Universität Ilmenau, 2006.

[9]: Lutz, H., Prof. Dr.-Ing; Wendt, W., Prof. Dr.-Ing.: *Taschenbuch der Regelungstechnik mit MATLAB und Simulink*. 7., ergänzte Auflage. Frankfurt am Main: Wissenschaftlicher Verlag Harri Deutsch, 2007.

[10]: Watkins, C.J.C.H.; Dayan, P.: *Q-Learning*. Machine Learning 8, 1992, S.279-292.

Abbildungsverzeichnis

Anhang

Die im Rahmen dieser Arbeit erstellten Programme und Daten finden sich auf der beiliegenden CD wieder. Im Wurzelverzeichnis befinden sich die Arbeit selbst im .pdf-Format sowie die Unterordner „MATLAB“ und „dSPACE“. Im Ordner „MATLAB“ sind die erstellten Dateien zur Regelung hinterlegt. Dazu gehören die Simulink-Modell-Dateien „*RLS_KNN_reales_Pendel.mdl*“, „*RLS_KNN_reales_Pendel_komplett.mdl*“, „*RLS_Simulation.mdl*“, die im Ordner „MATLAB/Initialisierung“ abgelegten MATLAB-Script-Dateien („**.m*“) zur Inbetriebnahme der Modelle und zur Verwaltung der Gewichte „*Initialisierung.m*“, „*Gewichtsuebergabe.m*“, „*Gewichtsverwaltung.m*“ und einige Beispielanwendungen (Ordner „MATLAB/Beispiele“). Außerdem sind im Ordner „MATLAB/Gewichte“ die Gewichtsdateien „*Gewichte1.mat*“ und „*Gewichte2.mat*“ zu finden, welche die zur Regelung benötigten und im Rahmen dieser Arbeit erstellten Gewichtskombinationen beinhalten.

Im Ordner „dSPACE“ ist das zur Überwachung und Aufzeichnung der Zustandsverläufe am realen Versuchsaufbau benötigte Experiment unter dem Namen „*Pendel_RLS.cdx*“ gespeichert. Desweiteren kann man in diesem Ordner die Layout-Datei „*layout_pendel_rls_expert.lay*“ finden.

Eidesstattliche Erklärung,

Hiermit erkläre ich, dass ich die am heutigen Tag eingereichte Diplomarbeit selbstständig verfasst und ausschließlich die angegebenen Quellen und Hilfsmittel verwendet habe.

Ilmenau, 01.06.2010 Andreas Reuter

Thesenpapier

I. Algorithmen des Reinforcement Learning können dazu verwendet werden Regelungsaufgaben quasi ohne a priori-Informationen den Prozess betreffend zu lösen.

II. Die Abbildung der Zustandswerte und Aktionen des Reinforcement Learning Agenten mit Hilfe künstlicher neuronaler Netze ist empfehlenswert, da diese die Fähigkeit zum Generalisieren und zur effektiven Speicherung von Daten besitzen.

III. Um das Verfahren am realen Versuchsaufbau anwenden zu können, müssen die Gewichte der künstlichen neuronalen Netze simulativ angelernt werden.

IV. Es können Parameter gefunden werden, die die Stabilisierung des Pendels ermöglichen. Die Verfolgung von Sollwertsprüngen bezüglich des Wagens sowie die Einhaltung der Sollposition ist ebenfalls möglich und kann mit Hilfe verschiedener Skalierungsmethoden verbessert werden.

V. Der Einsatz dieser Parameter ist am realen Versuchsaufbau möglich. Die Güte der durch diese Anwendung erzielten Regelung hängt von den Gewichtskombinationen ab und kann stark variieren.

VI. Im Vergleich zur Zustandsregelung zeigen sich bezüglich der Stabilisierung des Pendels kaum Unterschiede. Die instabile Ruhelage kann mit relativ kleinen Winkelamplituden aufrecht erhalten werden.

VII. Die Einhaltung der Sollposition verläuft weniger harmonisch als bei der Zustandsregelung. Bei Sollwertsprüngen kann jedoch ein geringeres Überschwingen beobachtet werden.

VIII. Ohne ein Modell, das die am Versuchsaufbau auftretenden Reibungseffekte abbildet, sind die Algorithmen des Reinforcement Learning nicht dazu in der Lage diese Effekte zu beseitigen und somit eine harmonischere Regelung zu ermöglichen. Desweiteren muss dem Agenten ein neues Ziel vorgegeben werden, um diese Effekte zu beseitigen/vermindern.